# Real-Time Cache Design

By

Hon-Kai, Cheung

Supervised By :

Dr. Chi-hung, Chi and Dr. Gilbert Young

Submitted to Department of Computer Science and Engineering

in partial fulfillment of the requirements for the

degree of Master of Philosophy

at

The Chinese University of Hong Kong

June 1996

# Abstract

With advances in computer technologies, real-time computing is becoming an important area, both for research and for system applications. As the processor speed is growing at an incredible rate, the bottleneck for overall system performance will depend on the speed of the memory system. As a matter of fact, cache which is a fast but small memory buffer is often used to bridge the gap between the processor speed and the main memory speed. However, it is noticed that caches are absent in many real-time systems today. It is because the extremely efficient of cache memory will rise an unpredictable performance to real-time systems in which *Time* constraint is the most important parameters. Since the size of the cache is very small, it can hold only a small portion of program instruction and data. Thus the performance is somehow further limited.

In this research work, we propose a new cache architecture design which is more suitable for real-time computing systems. The main idea of the new design is to protect the cache content during preemption. Current cache architecture is designed mainly for general-purpose computing systems at which preemption seldomly occurs. It does not possess any ability to protect its content across preemption in real-time computing environments. Once the cache content is flushed or modified, time will be needed to establish the content again. A high cache miss ratio results and the overall system preformance will be degraded. To protect the cache contents during context switching, we divide the cache memory into partitions (*private and shared partition*) and allocate them to the tasks based on our partition allocation algorithm. In our scheme, once the private partition is assigned to the task, only that task has the right to access the partition. So, the cache ( partition ) contents will not be changed by some other tasks during preemption. As a result, predictable performance should be obtained in real-time cache-based computing systems.

In order to increase the cache hit ratio of the cache partitions, prefetching technique will be applied on our cache architecture. It is because cache prefetching in the general-purpose computing systems has been shown to be effective to improve to cache perfromance. Moreover, with the help of the prefetch buffer to prevent data pollution and write

buffer to hide the memory latency of write operation. The degree of predictability will further be increased.

# Acknowledgement

I would like to express my gratitude to my supervisors, Dr. Chi and Dr. Young for their sincere guide during my research work. Without their help, encourage and experience, this thesis could not have been completed. Also, I would like to thank Dr. Chin Lu, Dr. Tony Lee and Prof. Tang Zhizhong to spend their valuable time on marking my thesis and giving many suggestions.

During these two years, I have worked with many good friends. They are Chi-Sum Ho, Yung Chan, Keith Mak, Chi-Kwun Kan, Chong-Meng Lee, Alywin Yu, Sau-Ming Lau, Wai-Kit Chan, Phyllus Leung and Terry Lau. They always teach, help and encourage me lots of things. Without them, I am sure that I can not stand these two years in doing research work lonely.

Lastly, I must express my sincere gratitude to my parents, my two younger sisters, my younger brother, my uncle, Rocky and the most of all, my dear love, Man-Ni, for their supports and encouragements. Especially Man-Ni, she always cares about my health, concerns my work, and encourages me in all ways.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Overview

*In real-time systems, a late answer is a wrong answer. Predictability is the essential factor for embedded real-time system.* [Han89]

From the above statement, it is clearly to see that the tight timing requirements (*Deadlines*), the correctness and safety of the real-time system are very important. In fact, the primarily difference between real-time systems and general-purpose ones is the meaning of time. [BW89] said that there are many interpretations of the exact nature of a real-time systems. However, they all have in common the notion of response time which is the time taken for the system to generate output from some associated input. Therefore, the correctness of a real-time system depends on both the logical result of the computation and the time at which the results are produced.

Actually, cache memory is essential to real-time system in order to provide the high throughput and performance needed by time-critical applications [SP95]. However, modern cache architectures are mostly designed for general-purpose computing systems where timing usually is not a constraint. In addition, modern processor implementation techniques, such as caches and pipelines, only address performance by lowering average clock cycles per instruction (*CPI*) but ignore worst case [CS91]. In fact, worst-case execution times ( WCET ) play an important role to determine the schedulability of a task set and hence must be guaranteed to ensure system reliability. These guarantees can be achieved

by allocating much more time to the task than it is actually used [SP95]. No doubt, this would result in lower system utilization due to many CPU idle times. As a result, some real-time system would use these idle CPU times to run the non-critical tasks, e.g., aperiodic tasks.

In order to obtain the tight execution bounds, a task must be executed to determine its longest path of execution. This can be done by running the task in no cache environment. But the most difficulty is to determine the portion of memory references that are cache hit. Actually, the upper bound on execution time is obtained when the cache hit ratio is zero, that is either disable the cache memory or in no cache environment. Whereas to determine the lower bound, the system and the task set must be analyzed. Since there are great variations in program flow within each task, each execution path may have a very distinct memory access behavior resulting in very different cache hit rates [SP95].

Furthermore, the preemptable multi-tasking environment does greatly affect cache hit rates and thus affect the program execution times. It is because the associated cache lines of preempted task in the cache may be over-written by preempting task. Such environment would result in significant reductions in tasks' cached state once the tasks resume their execution. Also, we should know that the degree of tasks' cached state reduction may be difficult to determine because of variations in tasks' execution pattern and scheduling of tasks. So, if such cache architectures apply on real-time system where timing is the most concern, the unpredictable performance of the caches will yield little or no benefits to the system. In some cases, the system performance is even degraded. The following example is used to show the effect of preemption on cache hit ratio.

In Figure 1-1, normal execution for Task A is run uninterruptedly until it is finished and then the control is passed to Task B. In Figure 1-2, Task A is run for some times and then it is swapped out. Task B is swapped in to run until it is finished and then Task A resumes its execution. Since some of its associated cache lines in the cache are over-written by Task B, Task A experiences transit-reload and needs to re-fill the cache content again. The shaded region indicates that an additional execution times are introducted by cache misses while these cache lines are reloaded. Thus, the execution time of Task A

becomes longer when compared to normal execution in Figure 1.1.

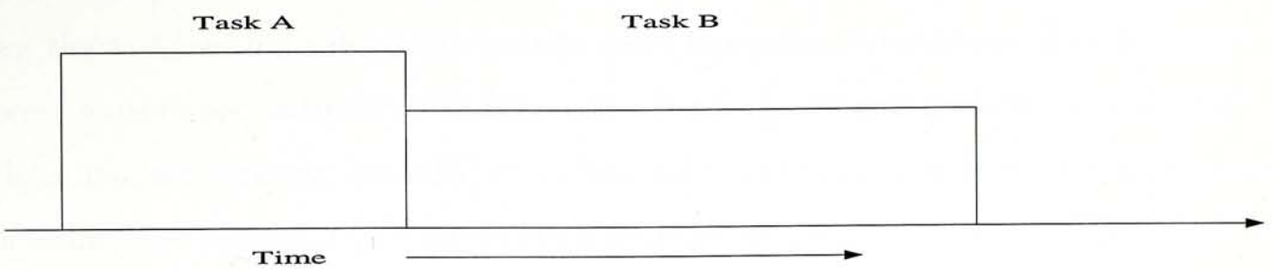


Figure 1-1 Normal Execution

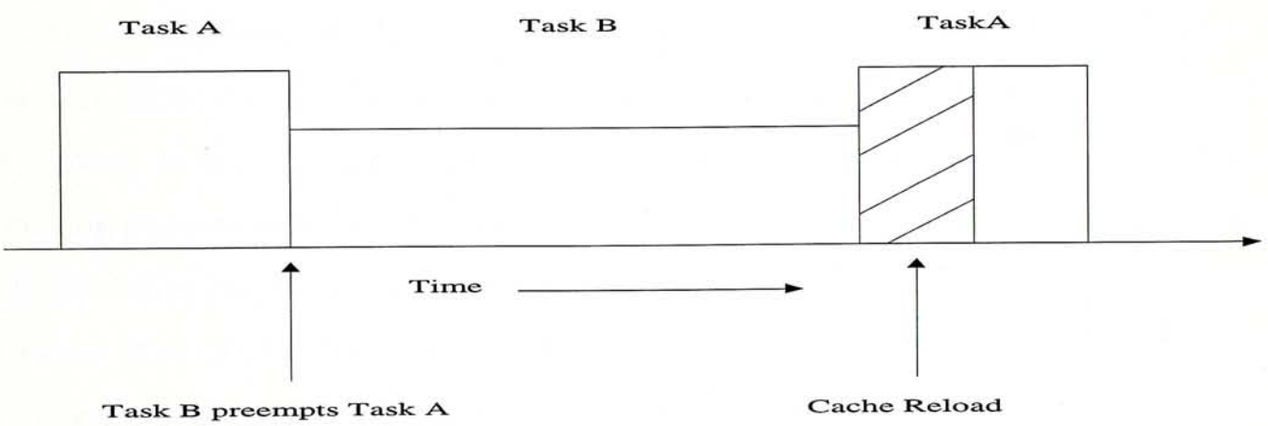


Figure 1-2 Execution With Preemption

Figure 1.1: Process Execution Between Preemption And Without Preemption System

In general, real-time systems are classified into two categories - hard and soft real-time systems. Hard real-time systems are those where it is absolutely imperative that responses occur within the specified deadline. Once the deadline is missed, the system is either halt or down. Soft real-time systems are those where responses times are important but the system still functions correctly if deadlines are occasionally missed. So, soft deadlines are usually modeled by a value function which decreases to zero gradually once the deadline is missed. Also, real-time systems are composed of periodic and aperiodic tasks ( processes ). Periodic tasks have regular arrival times whereas aperiodic task have random arrival times [KS90]. So, the ability to bound the amount of time required for a computation is essential to demonstrate the correctness of real-time systems [CS91].

In hard or soft real-time systems, the computer usually connects directly to some physical equipment and is dedicated to monitor or control the operation of that equipment. Also, the tasks within the system usually react to external events and data reported by process sensors and actuators. Today, such computing systems control many operations such as nuclear reactors, aircraft navigation, robotics systems, weapon systems, image-processing, and automated manufacturing facilities [KS90].

## 1.2 Scheduling In Real-time Systems

Since the timing requirement is the main concern in real-time systems, mechanism to make sure all or most of the tasks to meet such constraint becomes an important issue. In the past, in order to meet the deadlines of real-time systems, an overabundance of processing power and tasks scheduling using a cyclical executive system was used [KS90]. Such scheduling method in fact is a method that any changes either in the task set or in individual task can be devastating to the completed time line. Worst of all, correcting the time line may even result in a complete redesign of the time line sequence [LL73]. Hence, both system update and maintenance may be a disaster in a cyclical executive system. Sometimes, deadlines may be even missed because of transient-reload rather than the semantic importance of the task. Today, such methods has been replaced by some scientifically-based algorithmic scheduling techniques.

The scientifically-based scheduling techniques, either static or dynamic, have been proposed to guarantee that the system timing constraints are met. Two of them are the *rate monotonic* and the *earliest deadline* scheduling techniques.

The rate monotonic algorithm is a static scheduling technique. Under this algorithm, the priorities assigned to the periodic tasks are directly proportional to their time period [BW89]. The shorter the period, the higher the priority. In 1973, Liu and Layland[LL73] proved this algorithm can guarantee that n independent periodic tasks can be scheduled to meet all task deadlines if the sum of the task utilization for all n task is less than $n(2^{\frac{1}{n}}-1)$. Task utilization is defined as the task computation time divided by the task execution

time. For large n, the task utilization will be bounded to $ln\ 2$ or 0.69. However, this bound usually represents the worst case conditions. Kirk[Kir90] stated that in practice, the bound can be up to 90% or higher. It is because the task periods are often harmonic or nearly harmonic.

The earliest deadline algorithm is a dynamic scheduling technique. Under this algorithm, priorities are assigned to tasks according to the deadlines of their current requests. A task will be assigned with the highest priority if the deadline of its current request is the nearest, and will be assigned with the lowest priority if the deadline of its current request is the furthest. At any instant, the task with the highest priority will be executed [LL73], [BW89].

In summary, dynamic scheduling technique usually have task utilization bound of 100% or no processor idle time. However, it suffers from frequently run-time overhead and missed deadlines in an unpredictable fashion [SLR86], [LL73], [BW89]. On the other hand, static scheduling technique suffers from little run-time overhead but provide predictable performance.

## 1.3 Cache Memories

Cache memories are small, fast buffers which are placed between the CPU and main memory to bridge the speed gap due to very differ in their speeds. It is used to hold the most recently used instructions or data. Therefore, if memory references are served by cache, they will have a shorter access latency. Thus, the overall system performance can be improved. In fact, a typical instruction mixed-stream contains about 22% $LOAD$ instructions and about 5% $STORE$ instructions. So, about one-third of the whole instruction stream needs to access the main memory. Typically, a cache hit usually take one clock cycle whereas a cache miss may take 8 - 32 clock cycle [Cha95]. However, owing to the small cache size, only a small portion of the working set of instructions or data can be stored in the cache at one time. If the instruction or datum needed by the processor is not currently in cache, a cache miss occurs. That missed instruction or datum will be

retrieved from the main memory and put in the cache memories. In other words, the processor needs to access the next-level memory which usually leads to a longer access time.

In fact, the basic working principle of cache memory is taking advantages of two program localities, which are *temporal locality* and *spatial locality*. *Temporal locality* means that if an item is referenced, it will be reference again soon. This time related behavior can be loops and repeated procedure calls. *spatial locality* mean that if an item is referenced, nearby items will tend to be referenced soon [HP90]. This space related behavior can be a region of sequential code, or the access of sequentially stored data structure. Of course, if the programs have high localities, a good cache performance may be observed. Otherwise poor cache performance may be observed for programs with low localities.

In order to take advantage of temporal locality, a cache should keep more recently accessed instructions and data closer to the processor. On the other hand, to take advantage of spatial locality, a cache should have a block/line size larger than one word. It is because usually the larger the block size , the lower the cache miss ratio. Moreover, the efficiency of the cache can be improved by reducing the amount of tag storage relative to the amount of data storage in the cache [PP94]. Although a larger block size can decrease the cache miss ratio, at the same time, it can also increase the cache miss penalty. Since the larger the block size, the longer the access time from main memory. So, if the cache miss penalty increases linearly with the block size, larger blocks could easily lead to lower cache performance [PP94]. To avoid this situation, the bandwidth of main memory should be increased so that the transfer of cache blocks become more efficiently.

[HP90] indicated that cache misses can be classified into three types. They are *Compulsory*, *Capacity* and *Conflict* misses. *Compulsory*, also called *Cold Start*, miss refers to the first access to a block that is not in the cache, so such block needs to be brought into the cache *Capacity* misses are due to blocks being discarded and later retrieved since the cache cannot contain all the blocks needed during the execution of a program. *Conflict* misses occur when referenced data (instruction) is replaced by another data (instruction) mapped to the same set.

No doubt, the cache performance highly depends on various parameters such as cache size, set associativity, line size and so on. So, the degree of the above three misses can be reduced by choosing well studied values of these parameters. Unfortunately, these parameters are usually inversely related. Thus, trading off among these parameters to the best cache configuration has to be done with the help of simulations. In 1982, Simth[Smi82] described the detail of the effects of the above parameters on cache miss ratio in his paper. Furthermore, in 1991, he and his research fellows[JDGS91] preformed a complete cache miss ratio analysis on the entire set of SPEC benchmarks for different cache configurations.

In their simulations, [JDGS91] performed the experiments with different configurations on both unified cache and separated instruction/data cache. The cache size ranged from 2k to 1M bytes; the block size ranged from 16 bytes to 256 bytes; the set associativity was from direct-mapped to 8-way set associative. As shown in their report, *matrix300, nasa7, spice* and *tomcatv*, which were the floating-point programs in the SPEC benchmarks, gave the highest data cache miss ratios. However, when the cache size was increased to 512k bytes, the miss ratio for *nasa7* is still 2% more than other benchmarks for all the tested configurations. We all know that cache miss ratio decrease as the cache size increases. So, the cache miss ratio for *nasa7* dropped to below 1% when the cache size was increased to 1M bytes in some configurations. However, with a large cache size, the cache access (search) time becomes much more costly and therefore destroys the idea of cache memory.

In Ho's thesis[Ho95], he mentioned that even if the cache configurations are optimal, such configurations may not give satisfactory cache performance in all cases. It is because the optimal configurations try to maximize the program localities so as to increase cache hit ratio. However, sometimes the access patterns are actually beyond the scope of localities such as *nasa7*. As a result, a good cache performance strongly depends on both the localities of the programs and the optimal cache configuration.

Also, Kirk[Kir90] mentioned that, in general, 16K bytes cache can provide hit rate above 90%, whereas 4K bytes cache can achieve hit rate approaching 90% for a range of trace programs that include database management, scientific application, and batch jobs in general-purpose computing systems. Furthermore, in 1988, Kirk[Kir88] has shown that

cache with as small as 256 words can achieve hit rate of 75%.

## 1.4 Outline Of The Dissertation

The outline of the rest of the thesis is as followings:

- Chapter 2 describes the previous related work on real-time cache design.
- Chapter 3 describes the cache problems in real-time systems and the motivations to solve it.
- Chapter 4 presents the newly designed real-time cache architecture in full details.
- Chapter 5 states the simulation environments such as the properties of simulation trace programs and cache parameters to perform the experiments.
- Chapter 6 will be the analytical explanations of the simulation results.
- Chapter 7 will be the conclusion of this research work. Future research direction will also be proposed.

# Chapter 2

# Related Work

## 2.1 Introduction

In real-time computing systems, it is very important to guarantee that tasks must be met for correct operation and must be finished before their deadlines. In fact, conventional cache memory designs mainly focus on the determination of tight upper bound on worst-case execution time (WCET) in general-purpose computing systems. So, it becomes hard to have a good prediction for a task in preemptable multi-tasking environment. As a result, many researchers have studied that issue to some extents and suggested some solutions. Prefetching is a way to improve the system performances in today's general-purpose computing systems. Such technique may work in real-time computing systems. Other related materials are also discussed.

## 2.2 Predictable Cache Designs

### 2.2.1 Locking Cache Lines Design

In this approach, the most frequently accessed instructions and/or data in the task will be loaded at *Initial Program Load* time in the cache[Smi82], [Kir88], [RKW94]. Once the cache has been loaded, it is never modified. Since the cache memory contains the most frequently access routines, it can guarantee some degree of predictable performance. Of

course, the cache hit ratio will be somehow limited because of the limited cache size. Cache performance looks good when the frequencies of some of the routines in the task set are much higher than others. In other words, the cache performance becomes directly proportional to the ratio of task frequencies.

### 2.2.2 Partially Dynamic And Static Cache Partition Allocation Design

In this approach, the cache memory is divided into a predetermined number of partitions by means of hardware design[ELX88]. When a task takes control of the CPU at the first time, it dynamically requests a certain number of cache partitions. The partition remains dedicated to the task until it finishes. If the task is swapped out in the middle of its execution, it will resume execution with the exact cache partition contents that it has built up before swapping. Thus, the task never experiences any reload-transients caused by preemption. However, it suffers from two major drawbacks. First, if internal cache coherence needs to be guaranteed, shared data structures cannot be kept in the cache anyway. Second, sometimes cache partitions may not be available when a task requests allocation of cache partitions at the first time. It is due to the dedicated property of the partitioning to the task. So, the requesting task needs to wait for enough cache partitions before it can start execution. As the request of cache partition(s) is dynamic at the beginning but becomes static during runtime, this may result in unpredictable cache performance and priority inversions. [Raj89] defined that priority inversion occurs when a high priority task is blocked by a lower priority task.

### 2.2.3 SMART (*Strategic Memory Allocation for Real Time*) Cache Design

Kirk[Kir90] proposed the SMART cache design in his research. In this design, a cache of size C is divided into S segments. The number of segments depend on the size of the segment inputted from the command line. These segments are then allocated to N tasks

in an active task set, where N may be smaller than, equal to, or greater than S. The task has the right to access its own partitions so that the cache content can be protected across preemption. A portion of the cache segments forms one partition which is called *Shared partition* whereas the rest of them forms the *private partition* that will be assigned to the performance-critical or periodic tasks. The shared partition is used to service aperiodic tasks, and interrupts and to provide the coherent caching of shared data structures, and other miscellaneous tasks.

SMART cache design restricts the number of cache partitions that a task owned must be in the power of 2. Therefore, the number of cache partitions will be doubled when a task has successfully gained the extra partitions. The main disadvantage is that when a task requests the extra partitions, it may not have the enough number of cache partitions that can be satisfied to the requested task. Also, sometimes the cache partitions that a task owns will be under-utilized.

## 2.3 Prefetching

### 2.3.1 Introduction

Prefetching is a technique to increase the cache performance by fetching instructions and data into the cache memory or prefetch buffers before they are going to be referenced[Che93]. Similar to cache, prefetching algorithms also take the advantages of program properties, that is temporal locality and spatial locality. Generally speaking, prefetching instruction is considered to be more effective than prefetching datum. It is due to the sequential property of the instruction stream. For data stream, we have to investigate and predict the access patterns far enough in advance, so that the required data can be preloaded in the cache when they are required.

Smith[Smi82] mentioned that there are three common ways to perform prefetching. They are *prefetch on hit*, *prefetch on miss*, and *always miss*. *Prefetch on hit* means that the prefetching action is started when there is a cache hit. *Prefetch on miss* means that the

prefetching action is started when there is a cache miss. *Always prefetch* means that the prefetching action is started when there is either a cache hit or a miss. In addition, conventional prefetching algorithms can be classified into either hardware support or software assisted.

### 2.3.2 Hardware Support Prefetching

The simplest hardware prefetching method is called *One Block Lookahead* (OBL), which prefetches the block address i+1 when the block address i is referenced[Smi82]. OBL is simple to implement and does not have an extra runtime overhead.

In Chen's thesis [Che93], he indicated that there are four kinds of prefetching access patterns. They are scalar, zero stride, constraint stride and irregular. He verified that most hardware support prefetching methods perform well in the case of constant stride access pattern. Hardware prefetching methods highly depend on spatial locality of instruction and data references. In reality, instruction references have extremely high sequentiality whereas for data stream, only array references with unity stride have high sequentiality. So, hardware prefetching methods usually perform well in sequential access of instructions and data references, but they often fail in other access types.

Sometimes, even if data references possess high sequentiality, the prefetching performance may not be good. For example, consider an array reference A[$i$] with loop index i, which is increased by 5 in each iteration. With the cache line size of 2 words, all references to A[$i$] will be missed even if OBL is used[Ho95].

### 2.3.3 Software Assisted Prefetching

By using software assisted prefetching, a special "prefetch" instruction is inserted into the program code at the compile time to initialize the prefetch action[Oma81], [Por89], [Che93], [Ho95]. Such "prefetch" instruction is similar to a load instruction but without a destination register. In general, a function unit is responsible for executing the "prefetch" instruction is needed and also the "prefetch" instruction needs to be supported by the

instruction set. The compiler will analyze the program statically and then insert the "prefetch" instructions into the appropriate positions of the program code.

### 2.3.4 Partial Cache Hit

When prefetching techniques are used, a common phenomenon called the *partial cache hit* will occur. Although the prefetch request for a target address has been issued, if such instruction or datum is still not ready in the cache, a cache miss will occur when that instruction or datum is referenced. In this situation, we cannot consider it either a cache hit or cache miss. It is because the miss penalty in this situation is in between the miss penalty of a cache hit and that of a cache miss. The reason of having partial cache hit is that the time between the prefetch action and the actual reference is not long enough to allow the transfer of the target address from main memory to cache memory.

### 2.3.5 Cache Pollution Problems

Prefetched instructions or data may flush out the current working set of instructions or data when they are using the same cache. This is called *cache pollution* and it might result in performance degradation. The prefetched instructions/data may even flush out each other before they are actually referenced. As a result, cache pollution not only destroys the cache localities, but sometimes also makes prefetching useless. Fortunately, schemes [KL91], [CBM+92] that deal with this problem have been proposed.

One of the proposed scheme is to use a prefetch buffer ( cache ) which stores the prefetched working set. When a instruction or datum is referenced, the corresponding line will be transferred from the prefetch buffer into the cache.

## 2.4 Cache Line Replacement Policies

Cache line replacement policy is an important issue in cache design. When a new line is brought into the cache memory and there is no empty space available, some replacement

policies[Das89] are used to select which line needed to be removed from cache. The common replacement policies are First-In-First-Out (FIFO) policy, Least-Recently-Used (LRU) policy, Not-Recently-Used (NRU) policy and Most-Recently-Used policy.

For FIFO policy, the cache line that has been in cache memory for the longest period of time will be chosen for replacing. The advantage of using FIFO is that no instructions or data regarding past references need be maintained. When using the LRU algorithm, the victim cache line will be the one which was least recently referenced. Hence, it does require knowledge about past references and require more hardwares to implement than FIFO. When using the NRU algorithm, the victim line will be the one that has not been referenced in the recent past. The *recent past* is defined during implementation. The MRU algorithm is the one which is opposite to the LRU algorithm. That is, a cache line which is the most recently referenced is to be chosen for replacing.

[Smi82] showed that, on the average, FIFO is found to yield a cache miss ratio which is about 12% higher than by using LRU. Although LRU implementation is more costly than that of FIFO, from the perspective of performance, the LRU algorithm is preferable. Also, the definition of *recent past* is different to different systems and people, so NRU algorithm is not widely used. MRU is widely used in the program code which contains many "backward" or "repeated" looping. Consider a repeated array reference A[$i$] with n entries or items, using a cache line size of n-1 entries, the references to A[$i$] will cause a higher cache miss if LRU method is used. It is because the least referenced cache line will always be replaced. However, such cache line in fact is the most likely referenced cache line in the very near future.

## 2.5 Main Memory Update Policies

When a word in cache is modified by the execution of a store instruction, when to update such a word in main memory will become important[Das89]. Basically, there are two options for updating the cache. They are *write through* and *write back*. Write through approach means that the information is written to both the block in the cache and to

the block in the main memory at the same time. Write back approach means that the information is written only to the block in cache. The modified cache block is written to the main memory only when it is replaced [HP90]. Both schemes have advantages and disadvantages, Hennessy and Patterson[HP90] described this issue in full details.

When the CPU must wait for the write operation to complete, the CPU is said to be in *write stall.* So, a write buffer/cache is proposed to store those modified lines which are replaced by the cache line replacement policy. The advantage of using write buffer is to allow the processor to proceed on after the write operation and thus no stall is needed. Without a write buffer, the processor needs to wait for the completion of every write operation because of data coherence between main memory and cache memory.

Sometimes, write stall can still occur even with the write buffer. It is due to the write miss. There are two methods to deal with such situation. They are *fetch-on-write* and *no-fetch-on-write.* With fetch-on-write, the block is loaded from the main memory into the cache which is similar to the case of read miss. With no-fetch-on-write, the block is modified in main memory and is not loaded into the cache.

All these above policies were studied carefully by Jouppi[Jou93].

## 2.6 Summaries

In summary, we can draw a conclusion from the above three predictable cache designs. The idea of designing a predictable cache in real-time computing system is that a portion of cache needs to be used to hold the tasks' working sets which should not be over-written by some tasks during preemption. In fact, the mentioned three cache designs are able to do so. However, they also suffer some weaknesses that we have described before. In our research project, we will follow their direction to design a newly real-time cache. However, this cache will be designed not to suffer from the weaknesses of the mentioned predictable cache designs.

# Chapter 3

# Problems And Motivations

## 3.1 Introduction

As introduced in chapter 1, we have indicated that in preemptive ( multi-tasking ) environment, it is very difficult to predict the cache performance. This lack of predictability causes the real-time community to lag behind the general purpose community in utilizing cache memories [KS90]. In this chapter, we will discuss why modern cache memory cannot perform and predict well in real-time systems and will present our motivations to tackle such problems.

## 3.2 Problems

### 3.2.1 Modern Cache Architecture Is Inappropriate For Real-time Systems

Modern cache architectures are found to be very successful in general purpose computing systems because such architectures can take advantages of the program properties of *temporal* and *spatial* localities of the programs. Kirk[Kir88] mentioned that general purpose computing systems are primarily concerned with the total system throughput, average response times and the fairness. So, the conventional cache is designed to optimize the average case cache performance which yields average memory access times. These average

access times is closed to cache memory access times. However, the real-time computing system is primarily concerned with ensuring that individual task execution times are predictable. It relies on the worst case cache performance to provide the necessary timing guarantees. Worst of all, there is a large gap existed between the average case and the worst case cache performance.

Unlike real-time computing system, general-purpose computing system usually is a sole-task system which runs only one job at a time but not multi-tasking. Sole-task computing environment is an environment which has few or seldom interrupts or preemptions. During preemption, the cache contents normally will be either flushed out or over-written by other process. This means that the cache memory usually does not possess any ability to keep its cache contents remain intact across preemption. Actually, under sole-task computing environment, a process usually executes without any preemption until it finishes, which was shown in figure 1.1. As a result, when cache memory is applied to real-time computing environment, the cache content become free to be over-written by other process after preemption, which is shown in figure 1.2.

When using modern cache memory in real-time system, both the cache hit ratio and program execution time may suffer from an adverse effect. Since modern cache memory does not have any ability to protect its cache content, how the cache memory take advantages of the localities of programs in multi-tasking environment is a challenging problem.

### 3.2.2 Intertask Interference: The Effects Of Preemption

As said before, predictability is the core of real-time computing systems. In situations like real-time environments, where interrupts and preemptive scheduling are allowed, modern cache memory (*which is not designed to protect the cache contents during preemption*) will suffer from a high cache miss rate caused by intertask interference.

Intertask interference is caused by task preemption. In cache-based real-time systems, the cache content will be modified (actually flushed) during each context switching. Thus,

compulsory misses or cold start occurs every time when a new task takes the control of the CPU. During the cold start, cache reload-transient occurs. So, the cache needs to load the newly activated working set resulting in a significantly lower cache hit ratio. The worst case occurs when the task is being swapped out every time but it has not yet built up its working set in cache. This is the main reason why modern cache architectures may provide little or even no benefits to real-time systems.

Figure 3.1 shows the cache hit ratio against time, at which 90% hit rate will be obtained at time = T2 but the task is swapped out at time = T1 with 60% hit ratio, where T1 << T2. In multi-tasking environments, the resulting effect is shown on Figure 3.2. From Figure 3.2, we can see that when using conventional cache memory, the real-time computing systems will suffer poor overall system performance. When a task, P1, starts its execution, it establishes its cache content by loading its active working set into the cache memory. However, P1 is preempted by other task, P2, during its cold start period at time = T1. When the preempting task, P2, starts its execution, the same situation might occur again. P2 will be preempted out by P3 during its cold start period at time = T2. As a result, unpredictable cache performance results with low cache hit ratio. This situation is somehow similar to the *thrashing* phenomenon in the main memory which is caused by page fault. Dasgupda[Das89] has described this situation in full details. Hence, if most of the processor time is needed to service cache miss and very little time is used to actual productive computation, the performance of the system degrades drastically.

As we mentioned above, during preemption, the footprint of the preempting task will overlay the footprint of the preempted task. Footprint is defined as the number of distinct cache lines touched by a program[ST86]. Therefore, even if these two program could have fitted in the cache simultaneously, the probability for conflict is still high. It is because the cache lines of each program are not strategically analysis to avoid overlay with each other In other words, there is a great chance for the preempting task to displace useful lines of the preempted task. When the preempted task resumes its execution, it will still suffer from a very high miss ratio since it must reload its working set in cache. At a result, the system needs more time to finish the execution. Furthermore, in the environment

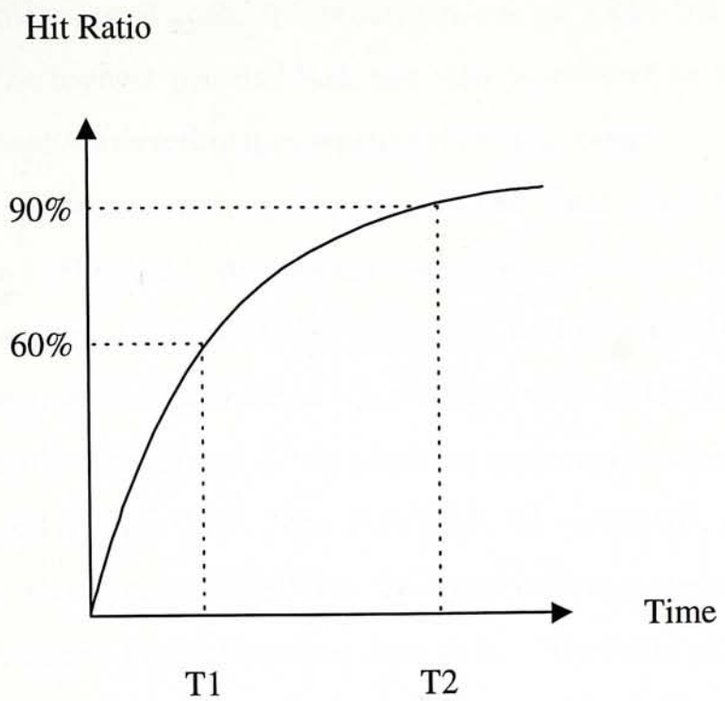


Figure 3.1: Hit Ratio Against Time Under Preemptive And Non-preemptive Environment

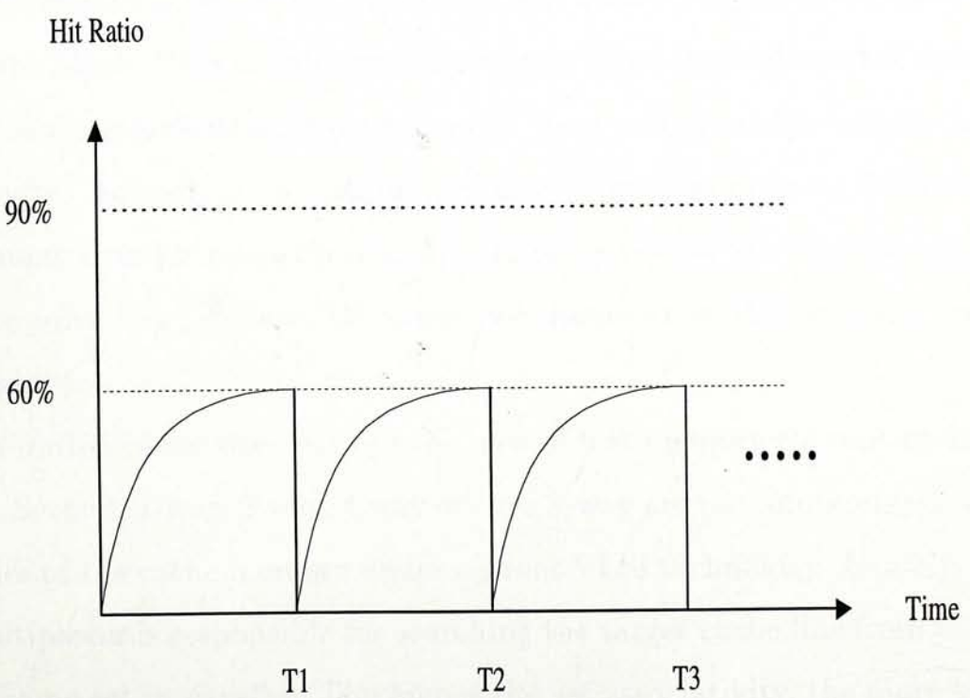


Figure 3.2: The Overall Effect of Preemptions to System Performance

of priority-driven preemptable schedulers, the low priority task may run to completion without any preemption, or it may be preempted one or more times by higher priority task. In fact, even the highest priority task can also be subject to reload-transients that are caused by interrupt service routines which utilize the cache.

Needless to say, context switching, which leads to *Cold Start* by flushing the cache content, is inevitable in the real-time systems. All these lead to be unpredictable behavior of conventional cache in real-time systems. The truth is that even if the real-time systems include cache memory, such cache memory is disabled most of the time. It is because the resulting performance improvement often leads to underutilization of system resources when cache is enabled at run-time. The AN/UYK-43 computer in the Navy's AEGIS combat System is a typical example[KS90]. This computing system consists of 32k-word cache which is partitioned for instructions and data. However, all task utilizations are calculated as if the cache is turned off because of the unpredictable cache performance. T

### 3.2.3 Intratask Interference: Cache Line Collision

Intratask interference occurs when more than one memory block of the same task map to the same cache block. This usually results in *capacity* misses and *conflict* misses. In order to avoid intratask interference, the cache must have unlimited size and/or having higher set associativity. Actually, it is also the problem in general purpose computing systems. Therefore, many frontier researchers and technology people are now spending their time and efforts to solve this problem. Of course, we should know that it is not easy to achieve such goal right now.

First, unlimited cache size destroys the idea of fast memory concept by increasing the access time. Second, 1-way, 2-way, 4-way or even 8-way are the common and acceptable set associativities of the cache memory under current VLSI technology. Usually, a unit which is called multiplexer is responsible for searching the target cache line from all of the cache lines in the same set in parallel. The higher the set associativity, the more the lines need to search in parallel. Hence, the multiplexer should be very fast and be able to perform

parallel searching. However, as the set associativity is increased further, the hardware cost will be too high even with recent VLSI technology. Third, since cache parameters are usually inversely inter-related, only studying cache size and set associativity may not be enough. Other cache parameters such as line size, cache replacement policy and so forth may also need to be studied as well.

In real-time systems, since the tasks must suffer from frequent context switching, not only the code in the same program but also the code of other programs compete for the same cache block. This would further devastate the problem of the intratask interference.

## 3.3 Motivations

### 3.3.1 Improvement Of The Cache Performance In Real-time Systems

As discussed in section 3.2.2, a cache is often disabled or not installed in real-time systems. So, the task set utilization is always poor. Remind that the definition of task set utilization[LL73] is defined as following,

$$Total\ task\ set\ utilization = \frac{Total\ task\ computation\ time}{Total\ task\ memory\ latency}$$

Remember that Liu and Layland[LL73] performed their experiments on no-cache or with cache-disabled real-time systems. However, in our project, we focus on the performance differences between partitioned cache and non-partitioned cache. Thus, we would like to define the meaning of task computation time and task memory latency.

- The task computation time is defined as the time of the task running with a cache which never misses.
- The task memory latency is defined as the time of the task running with cache on ( ie. with both cache hits and cache misses ).

In the ideal case, the task set utilization should be approaching 100%. That is, the task set is running with a cache which has minimum cache misses. On the other hand, the worst case should be a real-time computing environment with no cache or cache disabled. It is because the task set is running with 100% cache misses. With cache miss, the CPU needs to spend much more time to access the main memory than to access the cache memory.

The main objective in this research project is to increase the task set utilization. By using rate-monotonic scheduling algorithm, the lower bound of task set utilization to enable all deadlines to be met is about $n(2^{\frac{1}{n}} - 1)$. So, if the task set utilization can be improved, not only all or most of task deadlines can be met, but the total tasks execution times and the cache performance can also be improved. System and resource utilization will be increased because they are always underutilized at run-time when the cache is enabled[BW89], [KS90], [Kir90].

### 3.3.2 Hiding of Preemption Effects

In the above sections, we see that modern cache architecture does not have the ability to protect its cache content under multi-tasking environment. However, hardware venders can design a cache which has the ability to protect the cache contents by "freezing" the whole cache or a portion of the cache during preemption. From the hardware's points of view, once the size of the portion is defined, it may not be changed easily and so the feasibility is guestionable. A better way is to use software assists to protect the cache content during preemption.

Similar to *write stall* situation, a write buffer is used to store the modified cache lines temporarily so that the CPU will not be stalled on write operations. Thus the memory latency of write operations can be eliminated. Since cache content is modified during preemption, the key to achieve predicable cache performance is to hide the effects of the preemption and interrupts for all tasks presently in execution. Recently, there are two approaches to do so - *protection* or *restoration* [KS90].

*Protection* of cache content is a scheme which prevents the preempting tasks from destroying the cache information that belongs to the preempted tasks. The preempting task is not allowed to overlay information owned by the preempted task. Once the preempted task resumes its execution, the cache will appears undistributed by the preemption. In reality, the cache content has been changed, but not in the areas owned by the preempted task. Cache partitioning method is one of the protection approaches to ensure cache predictability in real-time computing systems.

*Restoration* of cache content is a scheme which allows the preempting task to overwrite the cache information that is owned by the preempted task. Before the preempted task resumes its execution, its previous cache will be reloaded so that the cache content will be exactly the same as before. However, we should notice that the overhead involved in restoring the cache might lead to a significant reduction in performance and thus could easily negate the benefits of a cache.

Figure 3.3 can help us to decide which approach, either protection or restoration, is most suitable in the real-time computing system.

When the cache size increases, the misses ratio will decrease. However, when the cache size is beyond a certain value, the misses ratio will become constant. It is due to the saturation of the cache with the current working set. Even with more cache space, the cache miss rate will not be decrease any more. Actually, when a task is allowed to run, it will utilize the cache space. The longer the time to run, the more the cache space will be utilized. In contrast, the shorter the time, the less cache space will be used. Since the tasks are now subject to preemption in real-time environment frequently, their execution times will be shorter when compared to that in general-purpose computing environment. As a result, even if we allocate a very large cache memory to the tasks, the tasks actually utilizes a small portion of it.

From Figure 3.3, We can see that in non-preemptive environment, the cache memory will reach its saturation point at time = Tj, with cache size = Cj and miss ratio = Mj. However, in preemptive environment, the cache memory will reach its saturation point earlier at time = Ti, with cache size = Ci and miss ratio = Mi. The explaination for this

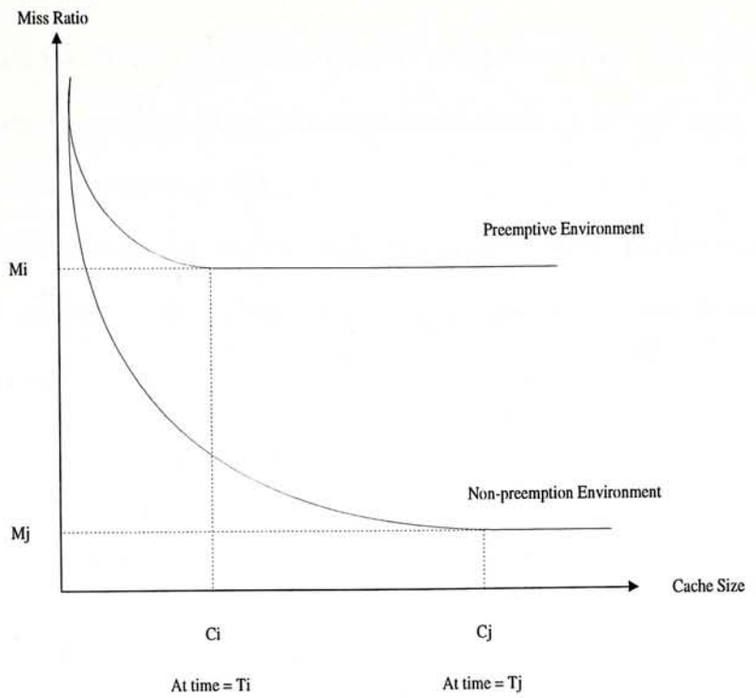


Figure 3.3: Cache Miss Ratio Versus Cache Size

phenomenom has been given above. Since the cache size Cj is much greater than that of Ci, the cache space Cj - Ci will be wasted in preemptive environment since this portion will not be used in this environment. Thus, the best way to do is to divide a large cache into many small cache partitions and then allocate the cache partition(s) to the tasks.

However, one may say that the miss ratio Mi is higher than Mj. As discussed before, in preemptive environment, a task will suffer from cache cold start once it resumes its execution after preemption. If we allocate such partitioned small caches to a task and make them to become private to that task, the task will only suffer cold start only once at the beginning. It is because the partitioned cache can only be accessed by the task that owns it. During preemption, by the private property of the partitioned cache, the cache content will remain intact since the preempting task has no right to access it. Also, more cache partitions can be allocated to tasks to lower the conflict and capacity misses. Hence, partitioning the cache could be the most suitable way to protect the cache contents in the real-time system during task preemption.

## 3.4 Conclusions

In conclusion, small partitioned caches which are private to certain tasks should provide better cache performance than large conventional cache ( which can be easily modified by preempting task during preemption ).

Judging from the above facts, we will propose a new *cache partitioning* method to hide the effects of preemption so that the cache become more effectiveness in real-time computing environment.

# Chapter 4

# Proposed Real-Time Cache Design

## 4.1 Introduction

In this chapter, we are going to propose our new real-time cache design in full details. Once again, the objectives of this research project are as follows:

1. Increase the overall cache hit ratio

2. Increase the task set utilization.

## 4.2 Concepts Definition

Before going into our proposed mechanism for real-time cache design, we would like to introduce some related concepts and definitions used in our proposed cache models.

### 4.2.1 Tasks Definition

As stated in chapter 1, the process which runs on the real-time computing systems is either periodic or aperiodic task ( process ). In this section, we will define the nature of periodic and aperiodic tasks that we assume in our proposed cache models.

1. A periodic task is defined as a task with periodic cycle execution and periodic deadline. The *deadline* for a task is defined to be the time of the next *request* for

the same task. That request is defined to be the cycle execution for a task. For example, if there is a periodic task, its cycle execution is 100 cycles and deadline is 300 cycles, then the task has to run at least 100 cycles in each 300 cycles. If the periodic task can meet this requirement, it is considered to be successful. Otherwise, the periodic task is considered to miss its deadline.

2. An aperiodic task is defined as a task with aperiodic ( random ) cycle execution and aperiodic ( random ) deadline. Aperiodic or random means that the exact value of cycle execution and deadline are varying according to the run time environment. For example, consider a system with two tasks, one a is periodic task ( P1 ) and the other one is an aperodic task ( AP1 ). If P1 has 100 cycles execution time and 300 cycles deadline period, the cycle execution time of AP1 will be 200 cycles and the deadline period will be 200 cycles. It is because after P1 is executed for 100 cycles, the next request for P1 will be 200 cycles later. During this period, the CPU will become idle. Thus, it is decided that *the CPU will use this idle time to execute an aperiodic task.* The longer the CPU idle time, the longer the cycle execution time and deadline period for an aperiodic task.

Now, consider a system with 3 tasks, two of them are periodic ( P1 and P2 ) and the other one is aperiodic ( AP1 ). If P1 has 100 cycles execution time and 300 cycles deadline period. P2 has 200 cycles execution time and 300 cycles deadline period. Then the cycle execution time and the deadline period of AP1 are both 0. This means that AP1 is idle and waits until the CPU is free. In this example, AP1 can get control of the CPU only after one of the periodic tasks finishes its execution. It is because the CPU now has some idle time to run the aperiodic task.

### 4.2.2 Cache Performance Values

In later section, we will introduce the core of the proposed real-time cache design which is the dynamic cache partitions re-allocation algorithm. This algorithm is based on the cache performance of each task. As a result, we would like to define two cache performance

values called *best-expected* and *standard-expected* cache performance. These two cache performance values will help us to make the decision on cache partition re-allocation. The *best-expected* cache performance value is used to find out a task which does not need to enter the competition of cache partitions. while the *standard-expected* cache performance value is used to find out a task which needs to be sacrificed by giving out its cache partition(s).

## 4.3 Issues Related To Proposed Real-Time Cache Design

In chapter 3, we already showed that in real-time computing environment, there is no need to allocate a large cache space to a task. It is because the task uses only a small portion of the cache and then it will be swapped out. The portion which has not been used is wasted. In order to fully utilize the system resources, the N-way partitioning method is used.

As shown in fig 4.1, the cache space is divided into N equal cache partitions and each cache partition will be allocated to a task by N-way partitioning ( NWP ) method, A cache partition can only be accessed by the task to which it is allocated. Furthermore, the content of the cache partition will be protected across preemption. As a result, a task which is preempted or swapped out in the middle of its execution can resume execution with the exact cache partition(s) content which it has established before the preemption.

In fact, when the number of cache partitions allocated to a task increases, cache miss rate should monotonically decrease because the overall cache size increases. However, we should notice that assigning private partitions does not eliminate cache miss completely but it makes the cache more predictable.

Once we divide the entire cache space into N equal cache partitions, we need to solve or handle some related issues in our proposed real-time cache mechanism. The issues are listed as following:

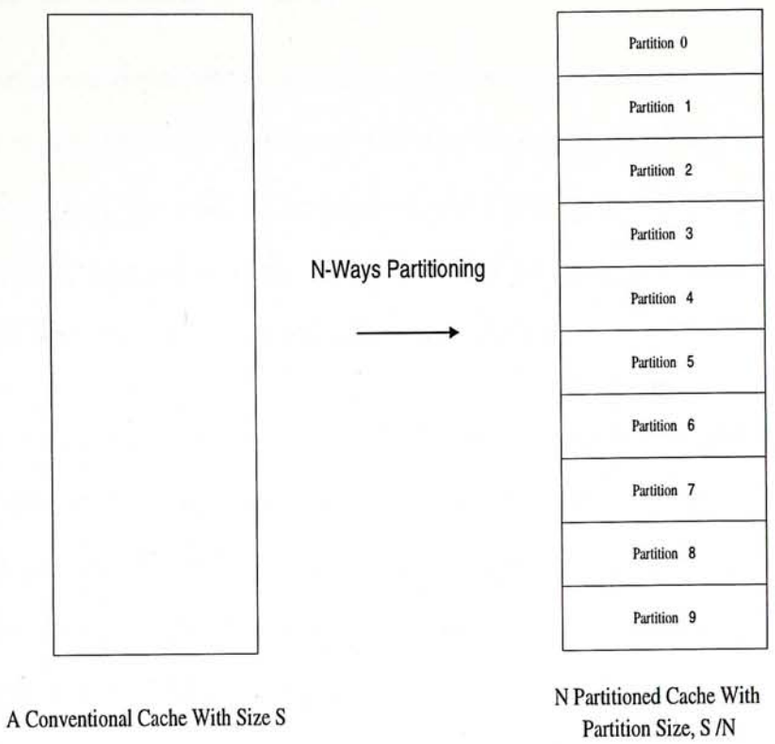


Figure 4.1: N-Ways Cache Partitioning

1. How are the periodic and aperiodic tasks served ?
2. How many of the private and shared cache partitions should be created ?
3. How are the cache partitions controlled ?
4. How are the cache partitions that a task owns reorganized ?
5. How is the bus bandwidth between the main memory and the proposed cache models handed ?
6. How are the proposed cache models addressed ?
7. How should the cache coherence in the proposed cache models be maintained ?

In the next few subsections, the above issues will be discussed one by one and the suggested solution will be then given. After that, we will go to the core of our real-time cache design.

### 4.3.1 A Task Serving Policy

In our proposed cache models, there are two queues to be constructed. They are *periodic task queue* and *aperiodic task queue* where the tasks are queued up and wait to be served. The higher the priority of the task, the higher the chance to be at the front of the queue. As the names suggest, periodic tasks will line up at the periodic task queue whereas aperiodic tasks will line up at the aperiodic task queue.

1. The periodic task serving policy is that when a periodic task, P1, is finished its cycle execution period, the control will pass to the next periodic task, P2, in the periodic task queue. It is because P2 now is at the beginning of the periodic task queue. At the same time, P1 will line up at the end of the queue and will wait to be served next time. Undoubtedly, the policy will try to serve the task during its cycle deadline period.

2. The aperodic task serving policy is that only when the first aperiodic task in the aperiodic task queue is completely finished itself, otherwise the control will not pass to the next one in the queue. Therefore, when an aperiodic task, AP1, is finished its cycle execution period, the control will not pass to next aperiodic task, AP2. In fact, AP1 is still at the beginning in the aperiodic task queue and hence will get the control of the CPU when the CPU becomes idle next time. AP2 will be at the beginning of the queue only when AP1 is completely finished.

Of these two queues, each task is associated with a time stamp so that we can keep track of its timing and check if the task meets its deadline or not. In next subsection, we will classify two types of cache partitions which are *private* and *shared* cache partitions. Periodic task can access either private or shared partition. Whereas an aperiodic task can only access shared partitions. However, when there is no task in the periodic task queue, all the cache partitions including the shared partition will become private cache partitions to that aperiodic task.

### 4.3.2 Number Of Private And Shared Cache Partitions

In our proposed cache models, they are two kinds of cache partitions. They are private partition and shared partition. Private cache partition is a cache partition which can only be accessed by its owner. So, after preemption, its cache content will not be changed by other tasks. A shared cache partition is a cache partition which can be accessed by any task. Its cache content is free to be modified during the preemption. With N-way partitioning, N cache partitions will be produced. Initially, all cache partitions are private cache partitions. But, shared partition will be created if needed.

After careful consideration, we decided that in our cache models there will be only one shared cache partition but N or N - 1 private cache partitions. The reasons for creating just one shared cache partitions are as follows:

1. If not all periodic tasks can own one or more private cache partitions, some of them will need to access the shared cache partition. As said before, the shared cache partition's content is free to be changed and hence might be unpredictable. That is why creating just one shared cache partition is better than having two or more shared cache partitions. One main purpose of our cache design is to decrease the degree of unpredictability. The more the number of private cache partitions available to the tasks, the higher the degree of predictability. Thus, having only one shared cache partition may be appropriate.

2. Since an aperiodic task only executes when the CPU becomes idle and it does not pass the control to the next aperiodic task until it is completely finished, one shared cache partition will be enough for the aperiodic tasks. It is better to allocate as many private cache partitions to periodic tasks as possible.

### 4.3.3 Controlling The Cache Partitions: Cache Partition Table And Process Info Table

In practice, a cache partition will be allocated to a task through the hash function or mapping function unit which is shown on fig 4.2. A *Cache Partition Table* ( CPT ) is used to keep track of the cache partitions that are assigned to a particular task, so we can know how many and which cache partitions a task owns. Each task bears an unique task ID to identify itself. In addition, the *CPT* uses several flags to determine whether a particular cache partition is a private or shared partition, and free or not free. The task ID is used as an index to search for its corresponding cache partition information. The structure of the *Cache Partition Table* is shown on fig 4.3. When a task is preempted out, the *CPT* will be updated.

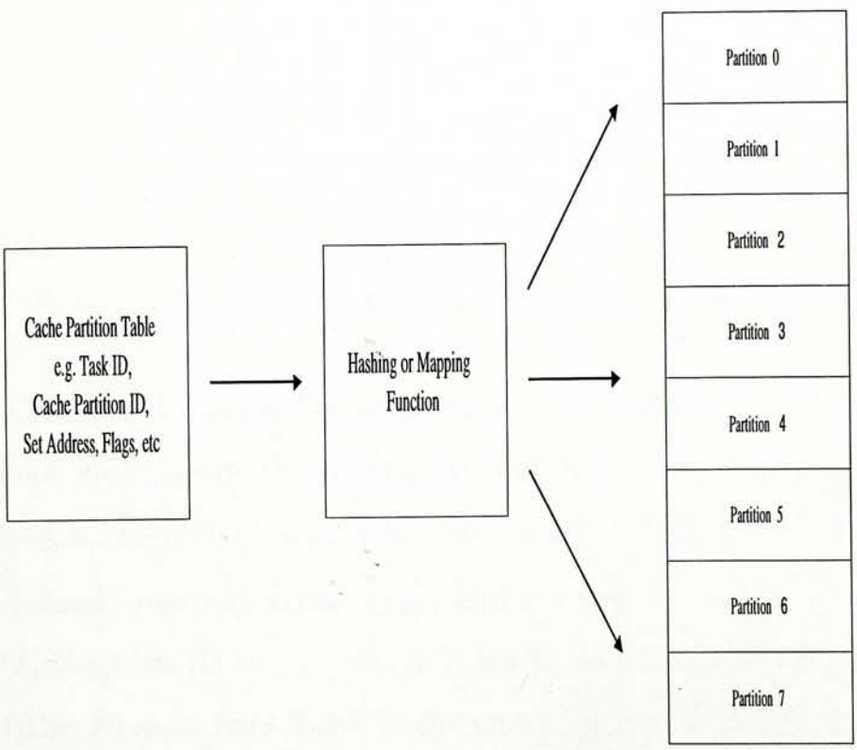


Figure 4.2: Controlling The Cache Partitions

During the preemption, when the preempting task takes control of the CPU, the preempting task ID will be used to find its cache partitions through the *CPT*. Hence,

the cache operations ( read or write ) can be performed as if it is a conventional non-partitioned cache, yet the cache content stored in the cache partitions can be protected during the preemption.

| Matching Task ID | Cache Partition ID | Flag to represent free partition | Flag to represent private partition |
|---|---|---|---|
| 3 | 0 | 0 | 1 |
| . | . | . | . |
| . | . | . | . |

Figure 4.3: Structure Of The Cache Partition Table

*Process Info Table* ( PIT ) is used to keep track of the statistics of all tasks during their execution. The statistics include the number of total instructions and data executed, the number of instructions and data cache hits, the number of different types of instructions accessed ( ALU, branch, memory access types and so on ), the task idle time and so forth. Same as the *CPT*, the task ID is used as an index to find its corresponding information. The structure of the *Process Info Table* is shown on fig 4.4. When a task is preempted out, the *PIT* needs to be updated.

The reason of having *PIT* is that the working principle of the dynamic cache partition re-allocation algorithm in our model will be based on the cache hit ratios of the tasks. Therefore, a mechanism is needed to keep track of the tasks' statistics.

| Task ID | Number Of Load Instruction | Number Of Store Instruction | Number Of ALU Instruction | Number Of Branch Instruction | Number Of Others Instruction | Number Of Data | Number Of Cache Hit | Number Of Cache Miss | The Task Idle Time |
|---|---|---|---|---|---|---|---|---|---|
| Task 1 | ' | ' | ' | ' | ' | ' | ' | ' | ' |
| ' ' | | | | | | | | | |
| Task M | | | | | | | | | |

Figure 4.4: Structure Of The Process Info Table

### 4.3.4 Re-organization Of Task Owns Cache Partition(s)

The number of cache partitions that a task owns can be either in the power of 2 or not. However, if we restrict the number to be in the power of 2, cache partition(s) sometimes are not fully utilized. For example, a task owns 4 cache partitions and if it needs one more cache partition to obtain the 90% or higher cache hit ratio. When the task requests the cache partition, in order to ensure the number of cache partitions to be in the power of 2, that task will be assigned 4 more cache partitions. After all, that task will own 8 cache partitions. However, since that task only needs 5 cache partitions, the extra 3 cache partitions will be under-utilized. This situation is similar to the case when the entire cache is allocated to the task. Moreover, sometimes the number of free partitions may not be enough to satisfy the need of the requesting task. As a result, the request is usually not granted and hence these partition(s) may be wasted.

In order to fully utilize all cache partitions, we decided the number of cache partitions

that a task owns needs not to be in the power of 2. That is, a task can own 3 cache partitions, 5 cache partition and so forth. Since the number of cache partitions is not restricted to the power of 2, we cannot re-organize a task's cache partitions in the way as shown in fig 4.5. In our proposed cache design, the task will re-organize its own cache partitions in the way as shown in fig 4.6.

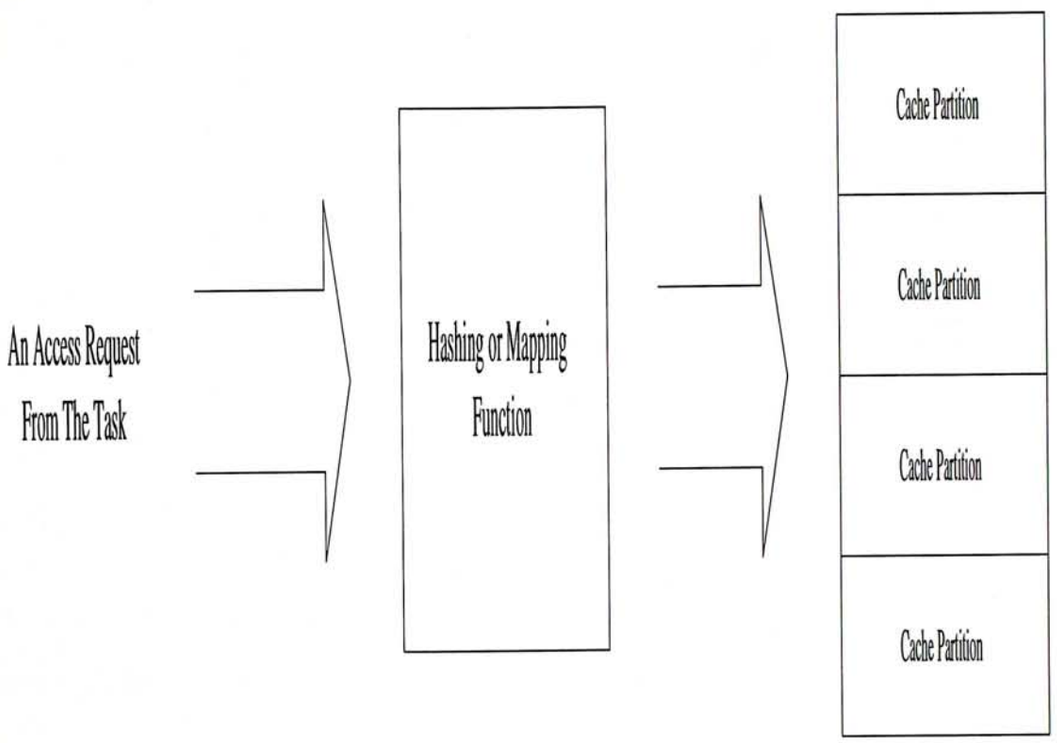


Figure 4.5: Re-organization Of Cache Partitions In The Power Of 2

### 4.3.5 Handling The Bus Bandwidth: Memory Requests Queue ( *MRQ* )

When the CPU fetches an instruction or data from the main memory, the CPU will send a read request to the main memory via its input/output system. The I/O system will serve all the memory requests. In our model, a *Memory Requests Queue* ( MRQ ) is constructed to serve all memory requests. They are five possible types of memory requests. They are Instruction Fetch Request (2), Data Fetch Request (2), Instruction Prefetch (1), Data

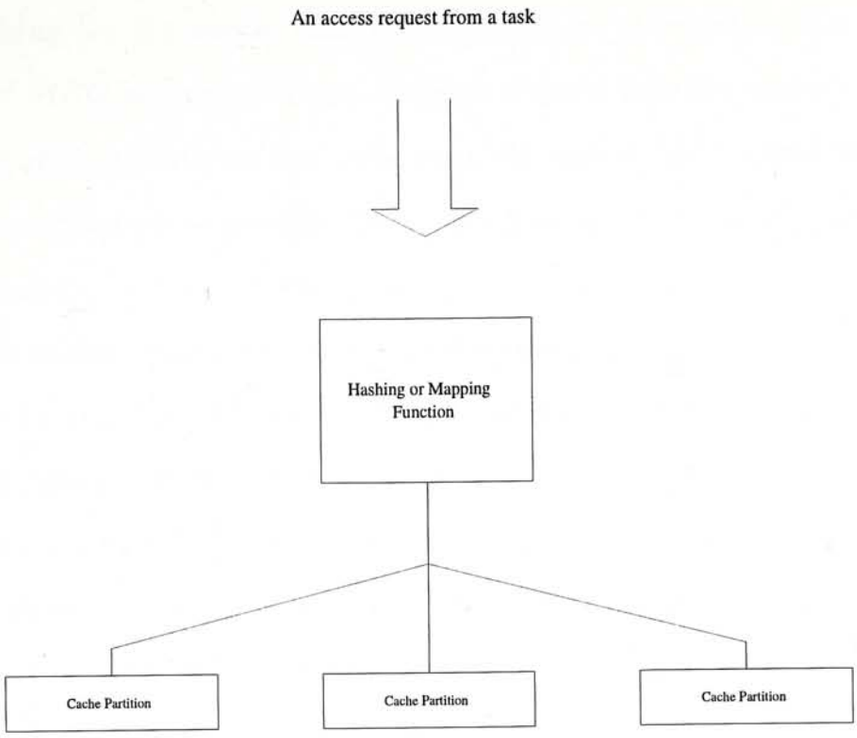


Figure 4.6: Re-organization Of Cache Partitions Not In The Power Of 2

Prefetch (1) and Write-Back Request (3). The number inside the parenthesis is the bus usage priority of that request. The larger the number, the higher the priority of the request.

The memory request queue is actually a small buffer which is used to hold the issued memory requests that have not yet been sent to the main memory. When the requests queue up at the *MRQ*, they are ordered by their priorities. For those that have the same priority, they are ordered by *First-Come-First-Serve*. When a request comes and finds that the queue is full, the sequence of the requests will be re-ordered based on their priorities. The last one is simply discarded if it is a prefetch request. Otherwise, the system will be stalled until the first three requests in the queue has been served.

### Prefetch Unit

All the prefetch operations are controlled by the prefetch unit. After receiving the appropriate matching signal and the prefetch address, the prefetch unit will initialize a prefetch

request by putting the prefetch address into the memory request queue. To avoid any unnecessary fetching for the same instruction or datum, a prefetch unit will check both the cache and the *MRQ* before putting a prefetch request into the queue. If the requested instruction or datum is already in the cache or in the queue, the request will be discarded. Always prefetch mechanism is used in the prefetch unit. That is, the prefetch request is issued regardless of cache hit or cache miss.

Under current cache design, *load/store* operations have priority over prefetch requests in using the bus bandwidth. When there is a cache miss, the CPU will be stalled until the desired cache line is fetched from the memory. In order to reduce the CPU stalling time, the desired cache line may be fetched from the memory in prior to all the prefetch requests. In our models, when either instruction/data fetch request occurs and there is a on-going request, the *MRQ* will perform either of the following operations:

- If the prefetch request has been served for more than half of the memory latency, then the on-going prefetch request will not be aborted and the fetch request for the desired data will be initialized immediately after the completion of the prefetch request, or
- If the prefetch request has **not** been served for more than half of the memory latency, then the on-going prefetch request will be aborted and the fetch request will be initialized immediately.

### 4.3.6 How To Address The Cache Models

In our proposed cache models, the least significant 2 bits ( bits 0 and 1 ) of every address are not used to select the word in the cache. Such two bits are used for byte-offset. The remaining bits of an address are divided into four fields to find the requested instruction or data. They are called T field, S1 field, S2 field and block-offset field. In fig 4.7, we use the SPARC architecture as an example to illustrate the addressing mechanism for our cache models.

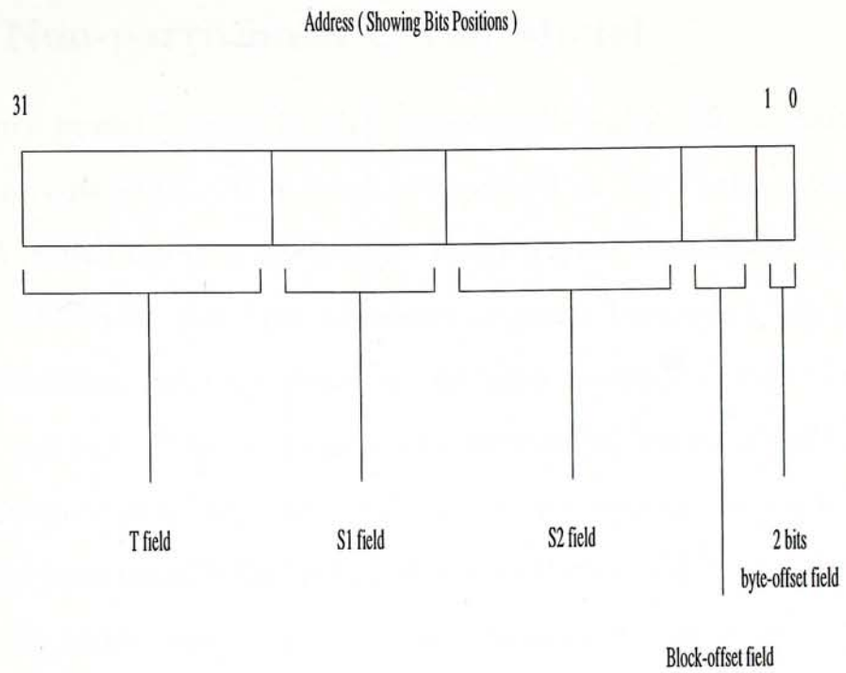


Figure 4.7: Bit Positions For Addressing The Cache Model

For partitioned cache, when a request is issued from a task, the system will use the S2 field as an index field to select the set in its cache partition(s). The T field and the S1 field will be grouped as the tag field of the requested address line and to compare the tag in the cache partition(s) in parallel. However, if the requested line cannot be found in its cache partition(s), the system will extend the searching to other tasks' cache partitions.

For non-partitioned cache, when a request is issued from a task, the system will group the S1 field and the S2 field as an index field to select the set in the entire cache. The T field will be used as the tag field of the requested address line and to compare the tag in the cache.

For both partitioned cache and non-partitioned cache, the system will the block-offset field to find out the desired word from the block or line. Since the tag comparison requires less time than to compare the whole address, the time needed for comparison is small enough to be neglected. Furthermore, we assume the time for parallel searching of the cache partitions is small enough to be neglected too.

### 4.3.7 Data Coherence Problems For Partitioned Cache Model And Non-partitioned Cache Model

The *data coherence property* requires that copies of the same information item at various memory levels are consistent. If a word is modified in the cache, copies of that word must be updated immediately or eventually in all higher levels[Hwa93]. Since the cache memory is now partitioned, the data coherence property becomes a big issue. If a word is modified in one partition, not only copies in the higher memory levels need to be updated but also those in the rest of the partitions. Hwang[Hwa93] gave a simple example to show the cache inconsistency problem when multiple private caches are used.

Consider a multiprocessor with two processors ( P1 and P2 ), each has a private cache and both share the same main memory. Initially, each processor reads the same line of information from the main memory into its cache. Then processor P1 writes new data into the cache after computation. No matter either *write-through policy* or *write-back policy* that P1 uses, the private cache of processor P2 will still contain the old data. So, if this line still remains valid, the ensuring access will return stale data. It is because these policies only ensure main memory coherence in multiprocessor environment[MDB88] but do not guarantee cache coherence in our partitioned cache.

#### Cache Line Placement And Replacement

In this subsection, we would like to discuss the cache line placement and replacement policy in our partitioned cache and non-partitioned cache. It is because we have to decide where to place the newly fetched cache line prior to solving the data coherence problem.

For the partitioned cache, when a task needs to fetch a new line from the main memory, the task will use the *cache partition table* to find all of its cache partition(s). Then, the system will use the index field ( S2 field ) of the requested address line to search the right set in a cache partition. After that, the system will first search for a free entry in this particular set in each of a task's partition(s). However, if there is no free entry in a set, the system will use the LRU policy to find the victim cache line. Once the chosen

cache line in each partition has been found, a signal will be sent together with the cache partition ID, the status of the chosen cache line and its time stamp to the mapping unit. The status of the chosen cache line is either *free* and *not free*. So, after receiving all the signals from the task's cache partition(s), the mapping unit will check to see if there exist a cache line with status *free* immediately. If yes, the mapping unit will put the requested address line in that cache line. But, when more than one cache lines are with the status *free*, the one with the smallest cache partition ID will be chosen to hold the newly cache line. In addition, when all the cache lines are with the status *not free*, then the one with the smallest time stamp will be chosen to hold the newly fetched line.

For the non-partitioned cache, when a task needs to fetch a new line from the main memory, the system will use the index field ( S1 field and S2 field ) of the requested address line to search the right set in a cache. After that, the system will search for a free entry in this particular set. Similar to the partitioned cache, when there is no free slot in a set, the system will use the LRU policy to find the victim cache line. Once the chosen cache line is found, the mapping unit will put the requested address line in that cache line.

For both the partitioned cache and the non-partitioned cache, if the chosen cache line is marked as dirty or modified, that dirty line will be put in the write buffer before being replaced by the newly fetched line. The dirty line(s) will be updated to the main memory only when the CPU becomes idle or no free entry for dirty line in the write buffer.

### Data Coherence Between Tasks' Cache Partition(s), Other Task's Cache Partition(s) And The Main Memory In Partitioned Cache Model

The working environment assumed in our study is a system with multiprogramming or multitask scheduling on a single processor. A task with either private cache partition(s) or shared cache partition, takes turn to control the CPU. When a process takes control of the CPU after preemption, its computation depends not only on the state of its owns cache partition(s) but also on the state of cache partitions owned by other tasks. This situation is similar to the situation in parallel processing environment. One of the approaches to

ensure cache coherence property is the *Illinois Cache Coherence Protocol*[PP84]. As a result, we would like to extend the basic idea of such protocol in our proposed cache model.

To maintain the cache coherence among a task's cache partitions, other task's cache partition and the main memory in our partitioned cache model is a tough job. Generally speaking, there are four possible states for cached blocks that we have to deal with. They are *Invalid*, *Valid-Exclusive* ( not modified and only one copy in only one cache partition ), *Shared* ( not modified but more than one copies in other cache partitions ) and *Dirty* ( modified and only one copy in only one cache partition ). When there is a read miss, a block will be fetched from the main memory into the cache. It is in the state of either *Valid-Exclusive* or *Shared*, depending on the presence of the block in other task's cache partitions. If the same valid block is not found in other cache partitions, then the the fetched block will be in *Valid-Exclusive* state. Otherwise, the fetched block will be in *Shared* state when there is more than one valid copy found in other task's cache partitions. A cache block in the *Dirty* state means that the local copy has been modified and the main memory has an obsolete copy. Furthermore, the *Shared* state indicates that the local copy is potentially shared in other task's cache partitions and all cached copies need to be identical to the main memory copy. In general, it is better to consider that the state *Invalid* includes the case where there is no valid block in the cache partition or the block is not present in all cache partitions.

The following is our proposed algorithm of the cache coherence protocol used in our partitioned cache models. Suppose a datum in a tasks' partitions Ci is accessed:

1. **Read Hit**. No coherence action needs to be taken.

2. **Read Miss**. If other cache partition Cj has a dirty copy, partition Cj will send the missing line to partition Ci and will also update the main memory at the same time. Both Ci's and Cj's copies will end up in *Shared* state. If there does not exist a dirty copy of the requested line but there exists *Shared* or *Valid-Exclusive* copies in other partitions, partition Ci will get the missing line from one of them and then all

cached copied will end up in *Shared* state. If there is no cached copy, the requested address line will be fetched from the main memory into cache partition Ci and will end up in *Valid-Exclusive*.

3. **Write Hit**. If the cache line in partition Ci is *Dirty*, no action is taken. If the line is *Valid-Exclusive*, its state will be changed to *Dirty*. If the cache line is *Shared*, all remote copies must be invalidated and Ci's copy will become *Dirty*.

4. **Write Miss**. Like a read miss except that all remote copies are invalidated and the cache line in Ci is set to *Dirty*.

As we all know, instructions do not have any coherence problem. But, we can still apply the idea of data coherence in this case. Normally, when a task experiences an instruction cache miss in its own cache memory, it needs to get it from the main memory. In our partitioned cache model, the task can still have a chance to find the target line in other tasks' cache partitions. Obviously, the time to get the line from other tasks' cache partitions is much faster than from the main memory.

### Data Coherence Between The Entire Cache And The Main Memory In Non-partitioned Cache Model

For a non-partitioned cache, since there is exclusively one copy of data in the cache, data coherence problem is just the matter between the cache memory and the main memory. Hence, using the write-through updating policy and the write buffer are enough to deal with the coherence problem.

## 4.4 Mechanism For Proposed Real-Time Cache Design

### 4.4.1 Basic Operation Of Proposed Real-Time Cache Design

Once the entire cache space is divided into N equal cache partitions, each of the cache partitions will be given a cache partition ID. At the same time, the operating system will assign the task priority to the tasks and put them into the task queue according to their task priorities. So, when a cache partition is assigned to the task, the task will store the cache partition ID in the *cache partition table*. Using the *cache partition table*, the task can find out which cache partition(s) can be legally accessed. Moreover, during the execution, the information such as the number of *load* and *store* instructions, number of cache hits and cache misses will be stored in the *process info table*. Also, the memory request queue will be used to serve both the fetch and prefetch requests. At the same time, the data cache coherence must be maintained as we have just described above.

During the tasks swapping, all the information of preempted task has to be updated in both *cache partition table* and *process info table*. When a preempting task gets the control of the CPU, it will use the *cache partition table* to find all its own cache partiton(s).

### 4.4.2 Assumptions And Rules

The core of the mechanism for real-time cache design is the dynamic re-allocation algorithm of each partition. This algorithm consists of two parts. They are the first round dynamic cache partition re-allocation, and the later round dynamic cache partition re-allocation. Actually, this algorithm is used to re-allocate the cache partitions to the tasks at runtime. Before introducing the dynamic cache partition re-allocation algorithm, we have to state several assumptions and rules.

- The basis unit size of one partition and the total cache size are in the power of 2.

- The number of cache partitions owned by one task is not restricted to the power of 2.
- The number of periodic tasks is noticed at the beginning and must be greater than 4.
- Cache Partitions re-allocation is performed when a "round" completes.
- A *round* is defined in any one of the following situations.
    1. Consider the periodic task queue is not empty but the aperiodic task queue is empty. A "round" means that when all the tasks in the periodic tasks queue has executed once and the control is passed to the first task in the periodic task queue again.
    2. Consider both the periodic and aperiodic task queue are not empty. After all the tasks in the periodic task queue has executed once, if the CPU has idle time, then the first task in the aperiodic task queue will be allowed to run. Finally, when the control is passed to the first task in the periodic task queue again, then this is called a "round".
    3. Consider the periodic task queue is empty but the aperiodic task queue is not empty. A "round" means that the control is passed from the first task to the next task in the aperiodic task queue.
- A task can only be assigned with one additional cache partition at each allocation round.

### 4.4.3 First Round Dynamic Cache Partition Re-allocation

During the initialization, the system clock is set to zero. The system clock has two fields. One is used to count the time of both cache hits and cache misses while the other is used to count the time of cache hits only. Also, each of N partitions will be pre-allocated to a periodic task ( out of M total periodic tasks ) according to their priorities which are given

by the operating system. Since we apply the *rate-monotonic* scheduling algorithm, the shorter the period the higher the priority. Of course, the higher priority task will have greater chance to gain the partition because of the first-come-first-serve property. Once all periodic tasks have gained their cache partitions or no more free cache partition can be assigned to the periodic tasks, the first periodic task in the periodic task queue will start to execute and the system clock will begin to count.

When either a periodic task which cannot be allocated a partition at the first round allocation or when an aperiodic task takes turn to control the CPU, a shared cache partition will be created dynamically and will then be reallocated to the task. In fact, such a shared partition is given by a victim periodic task with hit ratio that is the lowest among the N tasks ( those which has private partitions ). The system will refer to the information stored in the *process info table* to determine which task has the lowest hit ratio. After knowing the task ID, the system will use that task ID as an entry key to search the *cache partition table* and find out its corresponding cache partition. Then that private cache partition is marked as a shared cache partition. At the same time, the partition status flag and the matching task ID of such cache partition is updated to reflect that it is a shared cache partition now. After that the remaining tasks which do not own private partition can access this shared partition. Figure 4.8 is the pseudo-code of the the first round dynamic cache partition.

### 4.4.4 Later Round Dynamic Cache Partition Re-allocation

In the second and later round partition allocation, we will dynamically reallocate the partitions based on the cache performance of the tasks recorded in both the cache partition table and the process info table. During each round, the partition allocation procedure will do the following steps. These steps are called "Partition Pre-allocation Steps"

- If all periodic tasks has finished their executions, their cache partitions will be freed. Also, if there is an aperiodic task accessing the shared cache partition, such partition will become private to that aperiodic task. The cache partition table will

**Initialization**

```
system clock is set to zero;
for task ID = 1 to m periodic tasks {
        assign one partition to a task;
        number of free cache partition–;
        store the associated task ID in cache partition table and update the flags;
        if number of free partition = 0
            exit the initialization phase;
        }
```

**After initialization**

```
if ( running task = aperiodic ) or ( running task = periodic
and the private partition flag is false ) {
        if ( no shared cache partition ) {
            using the process info table to find out which task has the lowest cache hit ratio;
            change that task's private cache partition to shared cache partition;
}
        else
            access the shared cache partition;
            }
```

Figure 4.8: Algorithm For First Round Dynamic Cache Partition

be updated.

- If only one task ( which must be the periodic one ) uses the shared cache partition, the shared cache partition will become private to that periodic task and the cache partition table will be updated.
- If a finished periodic task owns some private partition(s), free up all its partition(s).

After pre-allocation procedure, we can know that how many free partitions is left for partition reallocation. To do the dynamic cache partition allocation procedure, there are only two possible situations: either N > 0 or N = 0.

**Number Of Free Partitions ( N ) > 0**

When N > 0, where N represents the number of free partitions, a partition will be allocated to *periodic tasks* based on their hit ratios in an ascending order. For instance, if there are 7 cache partitions and 5 periodic tasks. In the first round partition allocation, each of the 5 tasks gains one partition. So, in the second round partition allocation, there are 2 free cache partitions left. If the tasks hit ratios are 23%, 34%, 56%, 76% and 88%. Then the tasks with 23% and 34% will have chance to be assigned with one more partition. This is to increase the hit ratios of these 2 tasks. As another example, if there are 4 cache partitions and 5 periodic tasks. After the first round allocation, each of the 3 tasks will own one private cache partition and the other 2 tasks will share a shared cache partition. Sometime later, if a task which owns a private cache partition finishes, its cache partition will be free and will be allocated to the task which has the lowest cache hit ratio. Furthermore, that task should be one of the tasks which access the shared cache partition. As a result, 3 periodic tasks own private cache partitions and 1 periodic task accesses the shared cache partition. But, according to the Partition Pre-allocation Steps, since there is no aperiodic task and only 1 periodic task accesses the shared cache partition, that shared cache partition will be changed to private cache partition of that periodic task. Finally, the cache partition table will be updated and the dynamic cache partition allocation procedure will be exited.

However, if all periodic tasks has finished their execution, the free cache partitions will be allocated to the aperiodic task as its private cache partitions. Figure 4.9 is the pseudo-code when the number of free partition greater than zero.

```
i = 0;
if ( number of free partition > 0 ) and
( number of active periodic task > 0 ){
    sort the active periodic tasks based on their cache hit ratios;
    assign one cache partition to the sorted task
    i++;
    if ( number of free partition = 0 ) or
    ( i = number of active periodic task ) {
        exit the dynamic cache partition re-allocation procedure;
    }
else    if ( number of free partition > 0 ) and
    ( number of active periodic task = 0 ) and
    ( number of active aperiodic task > 0 ) {
        all the partitions will be allocated to that aperiodic task as private partitions
    }
}
```

Figure 4.9: Algorithm For Number Of Free Partition Greater Than Zero

**Number Of Free Partition ( N ) = 0**

When N = 0, there are three ways to reallocate partition(s):

- Check for the tasks which obtain cache hit ratios above the *best-expected cache performance value* from the process info table and also own more than one cache partitions from the cache partition table. If there exists one or more of this kind of periodic tasks, each one of them needs to give up one partition. We expect that even if the task loses one partition, it is likely to keep its hit ratio between the standard-expected and best-expected cache performance. At the same time, since N > 0 now, the free cache partition(s) can be allocated to those task(s) which have

lower cache hit ratios. The cache partition table will be updated and then exit the partition allocation procedure. Figure 4.10 shows the pseudo-code of that.

---

**Case I**

```
for i = 1 to m periodic tasks
    if ( the task cache hit ratio > best_expected_cache_performance )
    and ( that task's number of private cache partition > 1 ){
        free one cache partition;
        increment the number of free partition;
        update the cache partition table;
    }

back to the procedure for the number of free partition greater than zero
```

---

Figure 4.10: Case I For The Number Of Free Partition Equal To Zero

- If no free partition and no shared partition are shown in the cache partition table, more than one periodic tasks need to be sacrificed by giving out all of their private partitions. A shared cache partition will be created from one of them. For those tasks which has given out all of their private cache partitions, they now need to access the shared cache partition. The number of tasks that need to be sacrificed depends on the following two numbers.

  1. Number of tasks which obtain the standard-expected cache performance but without the best-expected cache performance, and
  2. Number of tasks which cannot obtain the standard-expected cache performance.

  From the process info table, it is easy to find these two values. In addition, we decided to choose the minimum value of these two numbers as the number of periodic tasks that need to give out their cache partitions. As said before, the task which accesses the shared cache partition will experience transient-reload after preemption

because the cache content in the shared partition is free to be over-written. So, we want fewer tasks to access the shared partition. Ideally, we want all periodic tasks can have their own cache partitions. However, if it is not possible to do, some tasks need to be sacrificed.

After careful consideration, the minimum number of periodic tasks that need to be sacrificed must be *greater than or equal to 2.* It is because if the number is 1, then only one task needs to give up its private cache partition. That private cache partition will then change to shared cache partition and will be allocated to the task which does not have any private cache partition(s). Clearly, the task which accesses the shared cache partition in this case is the same task which gives out this cache partition. So, if there is just one task accessing the shared cache partition, from the Partition Pre-allocation Steps, that shared cache partition will become a private cache partition to that task again.

After giving out the partitions by the victim tasks, we can free P or P-1 private cache partitions ( P-1 situation occurs if there is no shared partition created before ). Just like the case of $N > 0$, we will assign the free cache partitions to the tasks according to the ascending order of their cache hit ratios. However, there is a slightly modification this time. Only those tasks which still own private cache partition(s) can have the chance to gain the cache partition. After the allocation of free cache partition(s) to the tasks is finished, if there is one or some cache partitions left, they will be used in the next round dynamic cache partition allocation.

For example, given that the best-expected cache performance is 90% and the standard-expected performance is 50% hit rate. Assume that there is no free partition and no shared partition created and each task has only one partition with hit ratio is of 23%, 34%, 56%, 76% and 88% respectively. Now, three tasks meet the standard-expected cache performance, no task meets the best-expected cache performance, and 2 tasks below standard-expected performance. So, the number of victim tasks needed to free their private partition is 2. As a result, the task with 23% and 34%

will now give up their private cache partitions. One of these 2 partitions becomes a shared partition and the other one becomes free. Finally, that free partition will be assigned to the task based on the ascending order of the task's cache hit ratios. In this case, it is the task with 56% hit ratio. The two victim tasks will now access to shared partition. Finally, the cache partition table is updated and then the partition allocation procedure is existed. The pseudo-code of this algorithm is shown on fig 4.11.

- If there is no free partition but a shared partition is shown in from the cache partition table, the system will check whether there is at least 1 periodic task accessing the shared partition or not.

  - If there is at least 1 periodic task accessing the shared cache partition, the system will compare the cache performance between tasks with private partitions and with shared one according to the process info table. If a task using the shared partition "cannot" obtain hit ratio better than a task using the private one, then exit the partition allocation procedure immediately. Otherwise, the task which owns private partition(s) needs to pay the penalty.

    * If the task ( which suffers penalty ) owns more than one private partitions, it will give up one partition to that task ( which uses the shared partition ). Therefore, the task which uses the shared partition can now have private partition, or
    * If the task ( which suffers penalty ) owns only one private partition, it will give up its only private partition to that task ( which uses shared partition). So, the task which owns private partition will now need to use shared partition whereas the other task now owns private partition instead of using shared one.

  - If there is no periodic task accessing the shared cache partition, then check if all the periodic tasks whose cache hit ratios can reach the standard-expected cache

**Case II**

```
num1 and num2 are set to 0;
if ( number of free partition = 0 ) and ( shared cache partition flag is false ){
      for i = i to m periodic task {
          if ( task cache hit ratio > standard_expected_cache_performance )
          and ( task cache hit ratio < best_expected_cache_performance ){
              store that task ID in Temp1;
              num1++;
      }
          else {
              if ( task cache hit ratio ¡ standard_expected_cache_performance ){
                  store that task ID in Temp2;
                  num2++;
              }
          }
      }

if min(num1, num2) > 2 {
      if ( num1 > num2 ){
          free all the cache partitions with the task ID stored in Temp1;
          increment the number of free partition;
              }
      else {
          free all the cache partitions with the task ID stored in Temp2;
          increment the number of free partition;
              }
      create one shared cache partition;
      number of free partition−;
      the task which give out the private cache partition will access the shared partition;
      update the cache partition table;
      back to the procedure for the number of free partition greater than zero
          }
else
      exit the dynamic cache partition re-allocation procedure;
}
```

Figure 4.11: Case II For Number Of Free Partition Equal To Zero

performance. If yes, then exit the dynamic cache partition procedure. Otherwise, at most two periodic tasks whose cache hit ratios below the standard-expected cache performance need to sacrifice their private cache partitions. After giving out all of their private cache partitions, the task(s) will access the shared cache partition. As the number of free cache partitions now is greater than zero, the system will allocate a cache partition to the periodic tasks. Remember that only the one which still owns the private cache partition can have the chance to get the extra cache partition. The allocation is based on their cache hit ratios in the ascending order. If there is one or some cache partitions left after allocating the free cache partition(s) to the tasks, they can be used in next round dynamic cache partition allocation. The pseudo-code of this algorithm is shown on fig 4.12 and fig 4.13.

**Case III**

```
num1 and num2 are set to zero;
if ( number of free partition = 0 ) and ( shared cache partition flag is true ) {
      for i = 1 to m periodic tasks {
          if ( the task accesses the shared cache partition ){
              store that task ID in Temp1;
              num1++;
          }
          else
              store that task ID in Temp2;
      }

if ( num1 >= 1 ){
          compare between the task cache hit ratio with task ID stored in Temp1 and in Temp2
              if ( the cache hit ratio of the task with task ID stored in Temp1
              is less than that with task ID stored in Temp2 ) {
                  if ( that task's number of private cache partition > 1 )
                      give one cache partition to other task;
                  else
                  swap the cache partitions;
              }
          }
```

Figure 4.12: Case III For Number Of Free Partition Equal To Zero

```
else {
    for i = 1 to m periodic tasks
        if ( task cache hit ratio < standard_expected_cache_performance ){
            store that task ID in Temp1;
            num2++;
        }

    if ( num2 >= 2 ){
        free all the cache partitions with the task ID stored in Temp1;
        increment the number of free partition;
    the task which give out the private cache partition will access the shared partition;
    update the cache partition table;
    back to the procedure for the number of free partition greater than zero
        }
    else
        exit the dynamic cache partition re-allocation procedure;

    }
}
```

Figure 4.13: Case III For Number Of Free Partition Equal To Zero ( Con't )

# Chapter 5

# Simulation Environments

## 5.1 Proposed Architectural Model

To show the potentials of our partitioned caches, experiment on four different cache models under our defined real-time computing environment will be performed.

1. Model 1 is set to separated instruction and data cache with cache partitioning.
2. Model 2 is set to separated instruction and data cache without cache partitioning.
3. Model 3 is set to unified cache with cache partitioning.
4. Model 4 is set to unified cache without cache partitioning.

Remind that for Model 1 and Model 3, each task will utilize one or more private cache partitions instead of using the whole cache. However, for Model 2 and Model 4, all the tasks utilize the whole cache and transient reload of cache content will occur during the preemption.

Furthermore, a prefetch buffer and write buffer are implemented in these four cache models. The size of the prefetch buffer and write buffer are usually small, so items stored inside will be replaced very quickly.

For a prefetch buffer, *first-in-first-out* ( FIFO ) algorithm should be the most straight forward replacement policy to be used. However, in order to hold the most likely useful prefetched lines, we will use LRU algorithm. Also, the prefetch buffer is the read-only

buffer, so the data coherence problem between ordinary cache and prefetch buffer can be avoided.

For a write buffer, we use FIFO algorithm as its replacement policy. The set associativity for both prefetch buffer and write buffer is fully associative. That means the cache lines can map to any place of the buffer. A fetch-on-write policy is applied, so when there is a write miss, the desired address line will be fetched from the main memory to cache and will be marked as dirty.

Figure 5.1 to 5.6 are the logical flow charts of the Model 1 to Model 4.

## 5.2 Working Environment For Proposed Real-time Cache Models

### 5.2.1 Cost Model

In this section, we will define the cost ( time ) model of our simulation. Since we focus on the performance of the cache, for simplicity, we ignore the execution time for the ALU, Branch, and other types of instructions. In other words, we only focus on the latency introduced by the memory system:

*Time taken for accessing the cache = 1 cycle*

*Time taken for accessing the prefetch buffer = 1 cycle*

*Time taken for accessing the write buffer = 1 cycle*

*Time taken for accessing the main memory = C1 + C2 * ( N - 1 )* cycles

where *C1* is called the start up time, the time between sending out a memory request and receiving the first words from main memory; *C2* is the transfer time per words; $N$ is the number of consecutive words per cache line. In our simulations, we will consider the case of accessing the main memory as cache miss. While the other cases will be considered as cache hit.

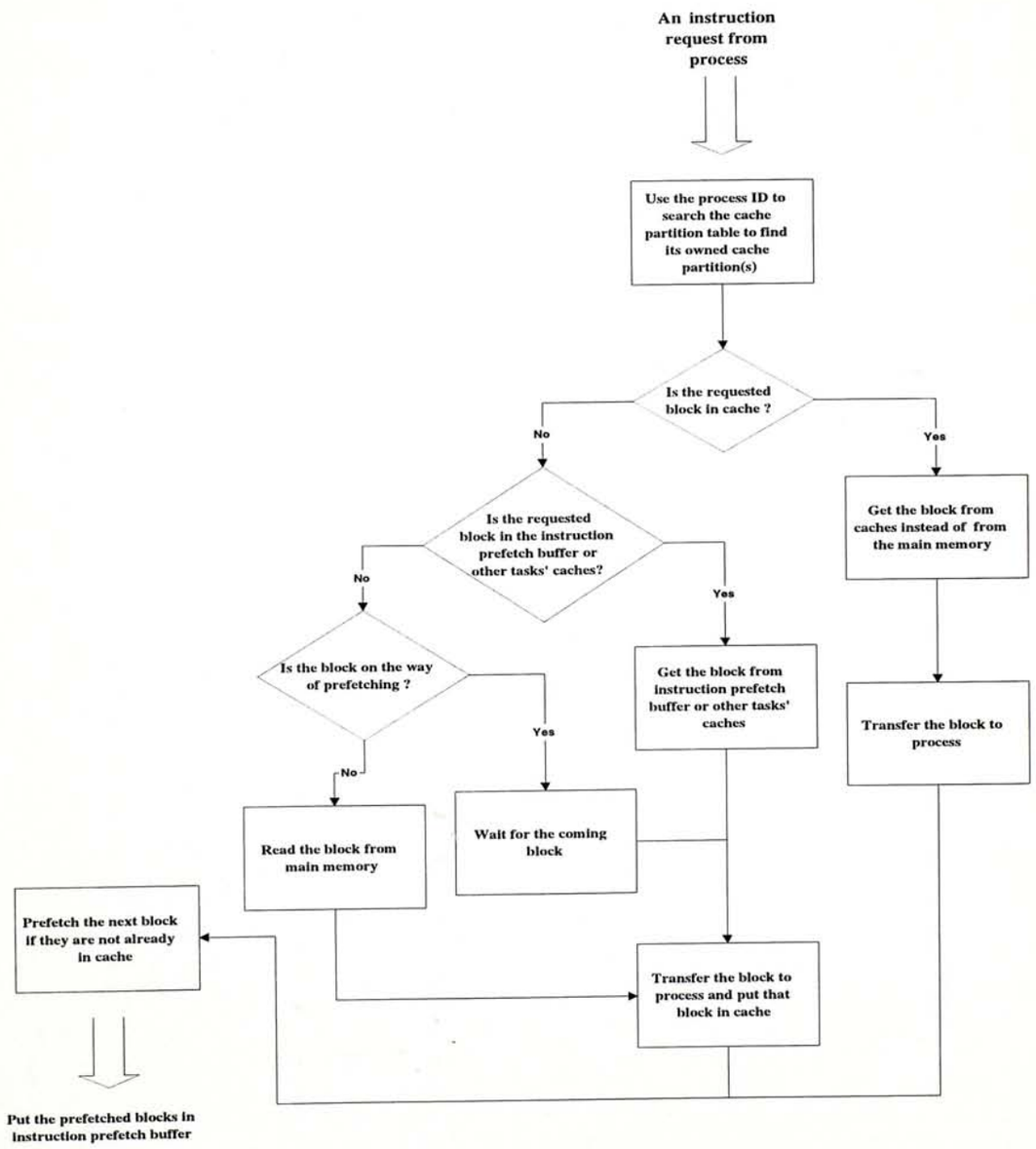


Figure 5.1: Instruction Reference Flow Chart Of Model 1

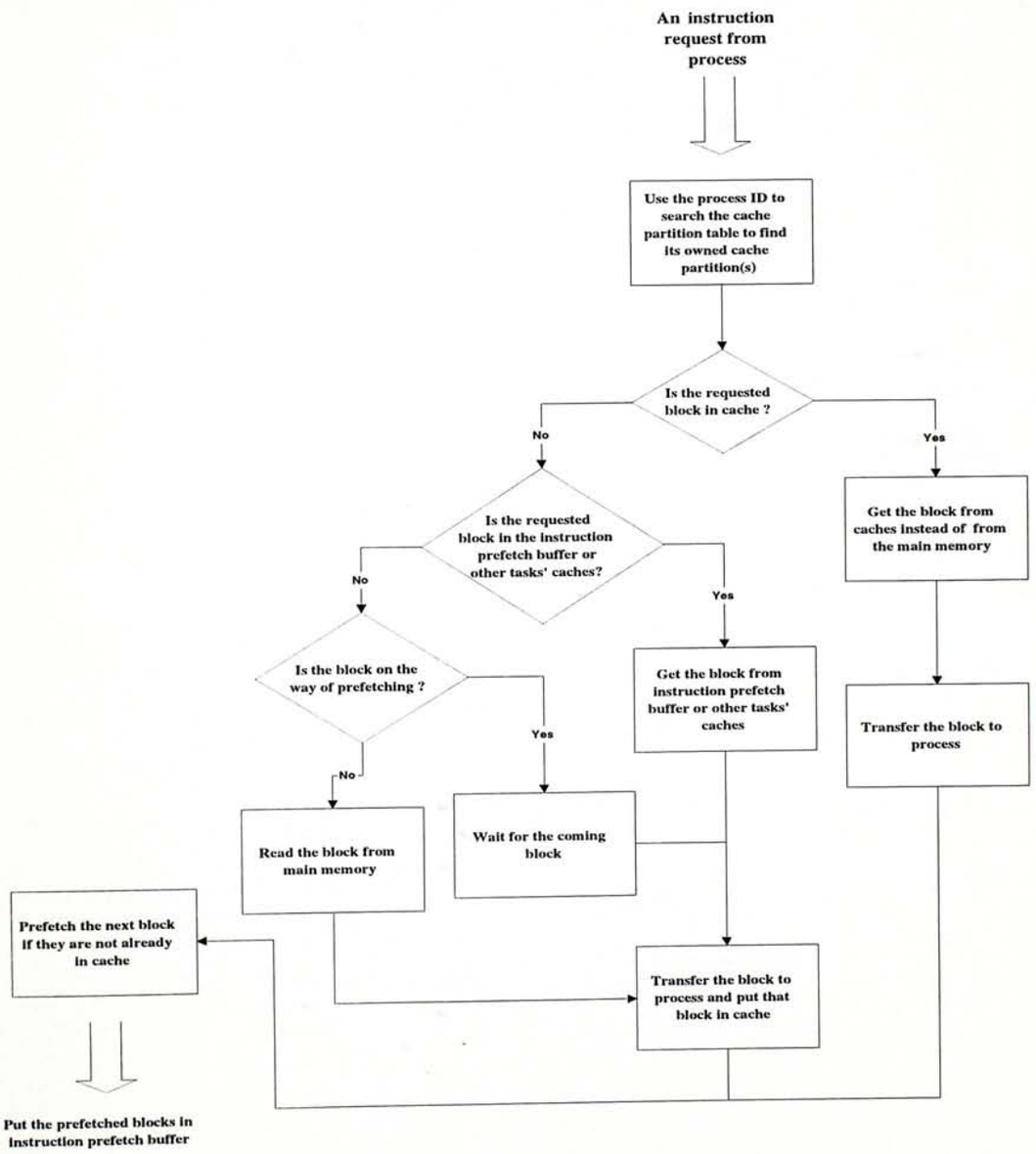


Figure 5.2: Data Reference Flow Chart Of Model 1

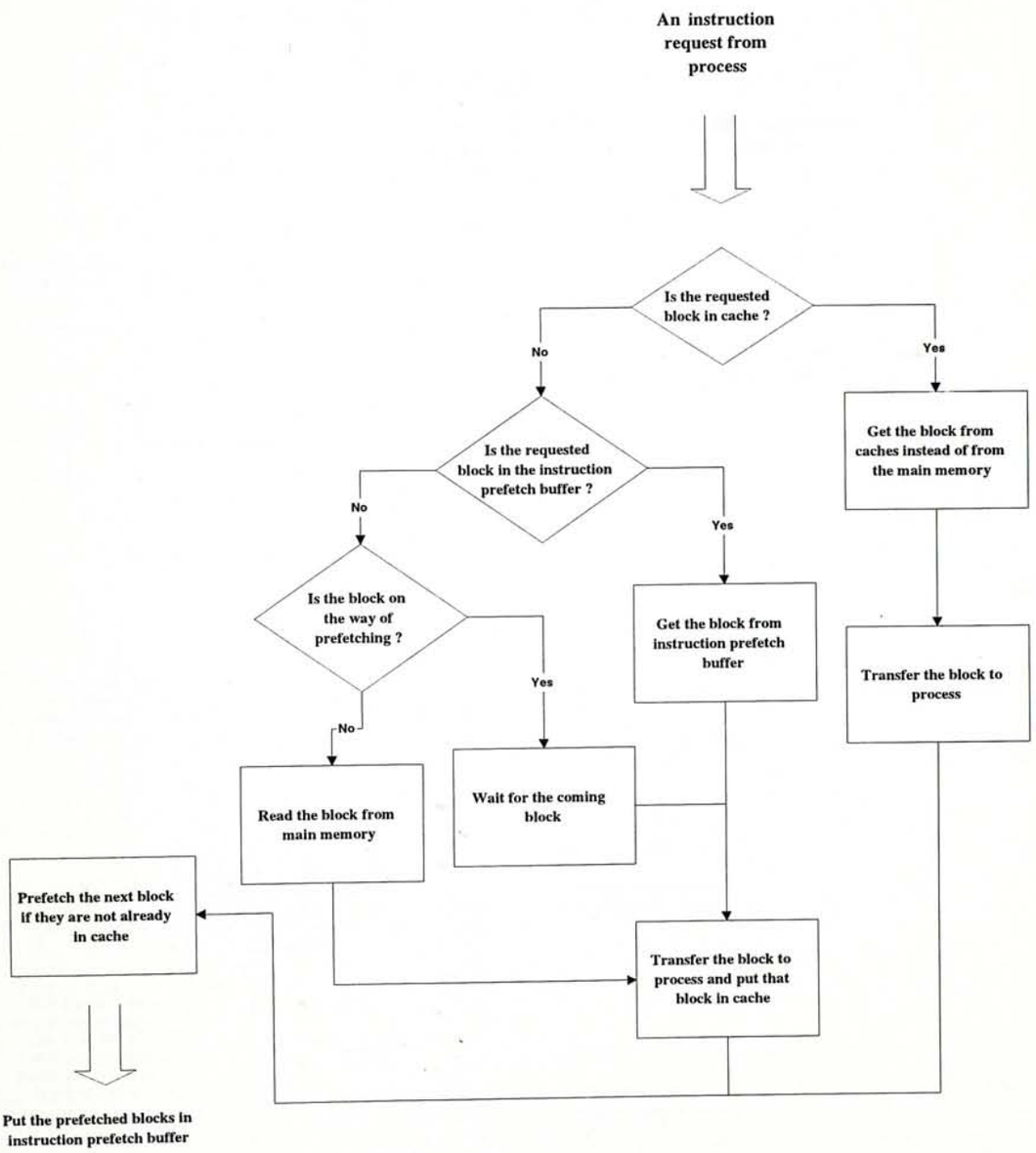


Figure 5.3: Instruction Reference Flow Chart Of Model 2

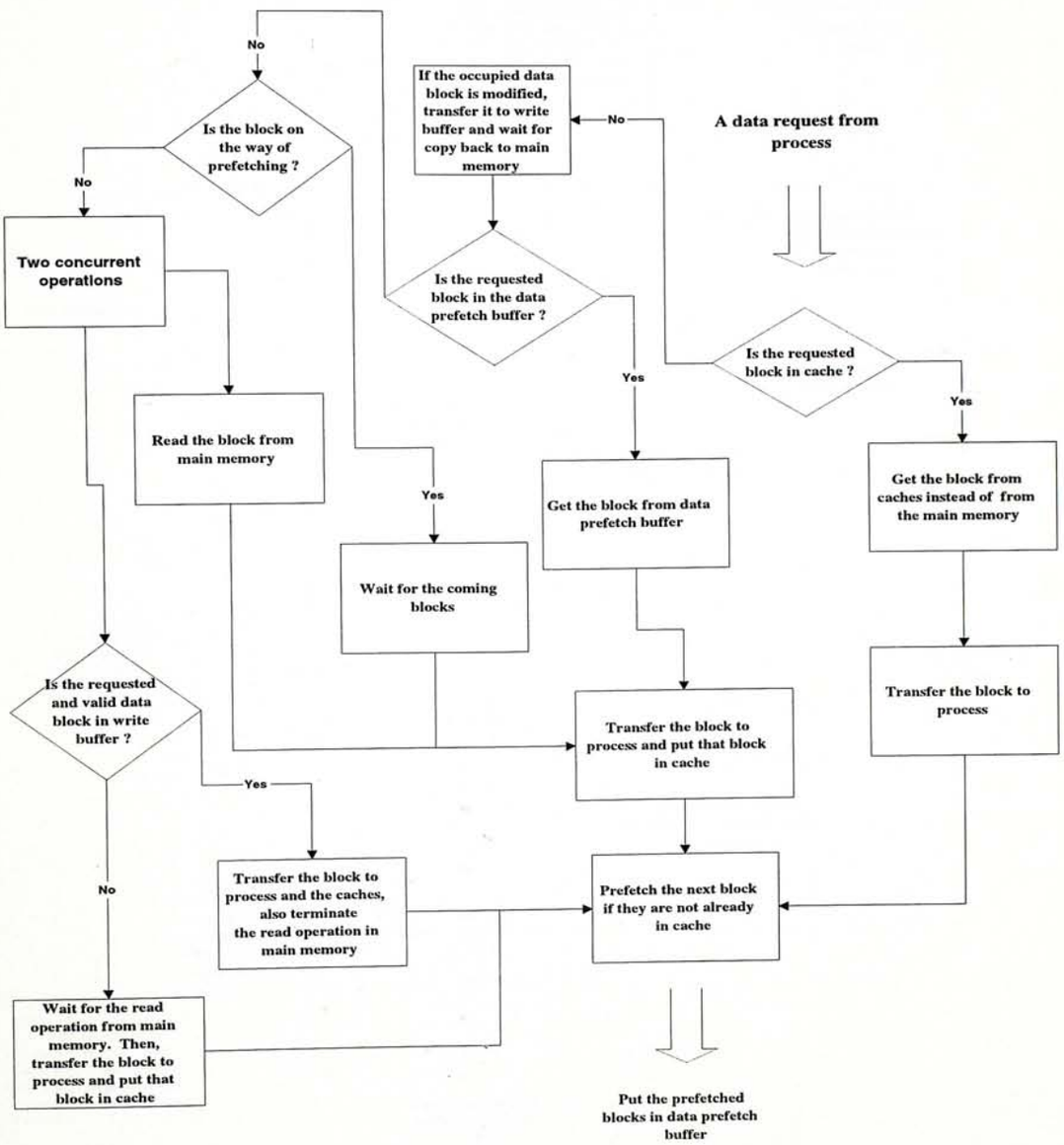


Figure 5.4: Data Reference Flow Chart Of Model 2

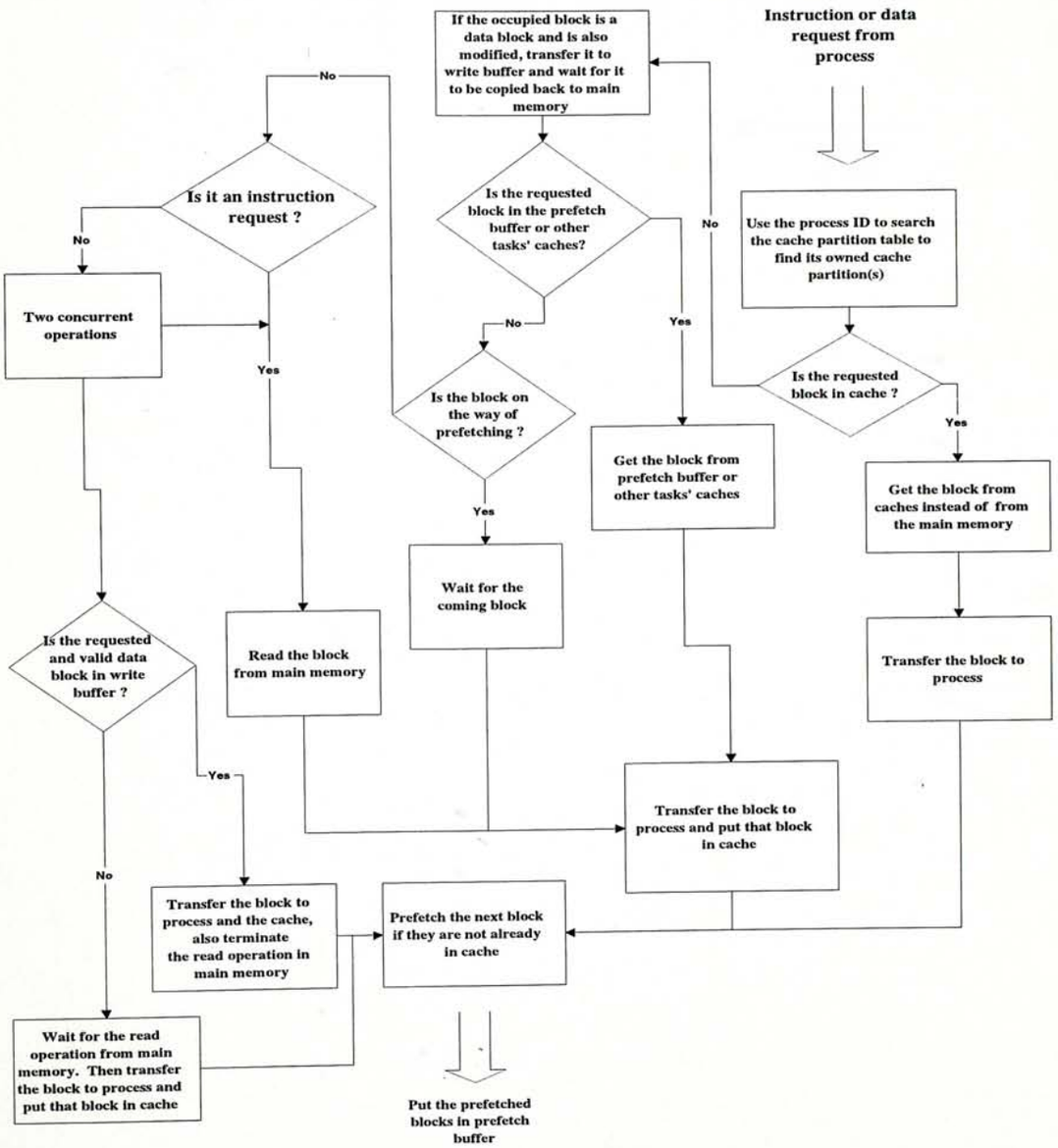


Figure 5.5: Instruction/Data Reference Flow Chart Of Model 3

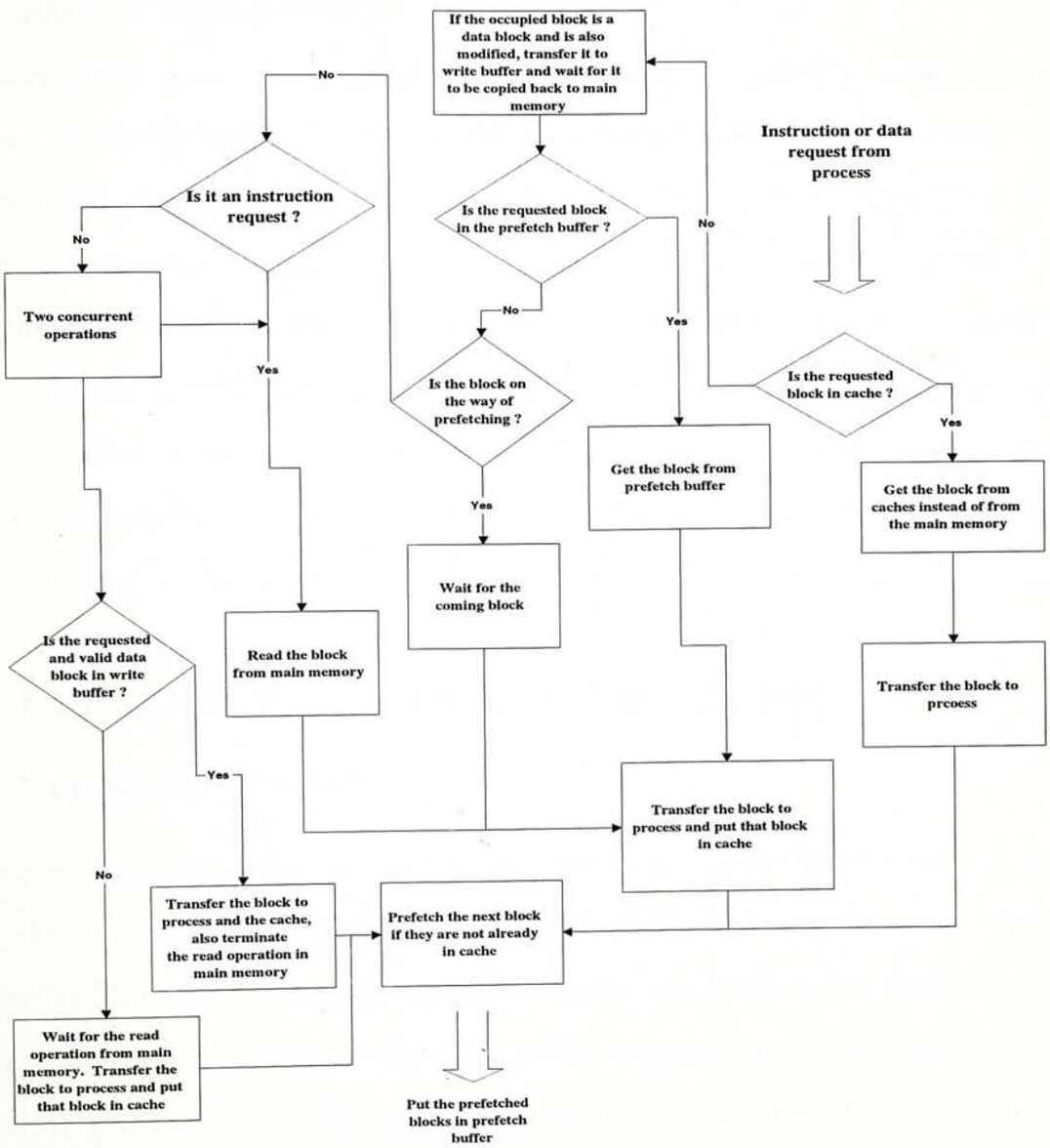


Figure 5.6: Instruction/Data Reference Flow Chart Of Model 4

### 5.2.2 System Model

In our simulation, we choose the **soft** real-time computing system as our simulation environment. The deadlines of both periodic and aperiodic tasks ( processes ) are **soft** deadlines. Therefore, even the task deadline is missed, the system can still function correctly instead of a system halt.

For example, if given that each of the two process, P1 and P2, needs to execute 100 cycles in a 200 cycles period and P1 is higher priority than P2. Due to cache misses, P1 may execute more than the pre-allocated 100 cycles. It is because when P1 has executed 99 cycles, it encounters a cache miss and cache miss penalty is $C1 + C2 * ( N - 1 )$ cycles. When P2 executes, it will miss its deadline definitely. As a result, **soft** real-time computing system is assumed in our simulation environment. Note that our models can easily be extended to **hard** real-time system. In this situation, additional and fixed time cycles which is equal to the cache miss penalty will be provided to each task so that they can have enough time to finish the fetch operation.

### 5.2.3 Fair Comparsion Between The Unified Cache And The Separate Caches

In our proposed cache models, since we have a *Memory Request Queue* to serve all memory requests, the bus bandwidth between the CPU and the caches is same for both between the *unified* cache and the *separate* caches. In order to obtain a fair comparison between the *unified* caches and the *separate* caches, the total cache size used should be the same. Therefore, a *separate* 1-KB instruction cache and 1-KB data cache should be compared with a 2-KB unified cache.

Moreover, we will construct an unified prefetch buffer and separate prefetch buffer for an unified cache model and separate cache model respectively. Still the number of entries in the unified prefetch buffer will be the sum of the number of entries in separate instruction prefetch buffer and data prefetch buffer.

### 5.2.4 Operations Within The Preemption

During preemption, all the cache contents in the shared cache partition, the prefetch buffer, the write buffer and the requests in the memory request queue will be flushed out and killed. However, there are some exceptions in our defined working environment.

First, if there are write back request(s) waiting in the memory request queue, they will be allowed to finish. This is to maintain the data consistency in cache. Of course, all other requests will be killed to allow the write back operation to start immediately. After that, the queue and the write back buffer will be flushed. Second, the valid and dirty data lines in the cache will not be updated. It is because it will take too much extra time to write back all dirty lines. In fact, only some or even none of these lines will be referenced. Therefore, the most appropriate way is not to touch those lines. If those dirty lines have to be updated, the write buffer and write through updating policy will overlap the write operation with some read operations. Theoretically, it will not increase the execution times too much for the preempting tasks when compared with updating all the dirty lines in one burst mode. Last of all, when all the tasks have finished execution, we will check if the caches still contain some valid and dirty lines. If yes, all the dirty lines will be written back. In our simulation, such time will be equally divided and then be added to the execution time of each task.

## 5.3 Benchmark Programs

In our simulations, all the trace files used are from the SPEC92 benchmark programs and will run on SPARC machines. Totally, we used 6 benchmark programs in our experiments. Four of them are floating benchmark programs - NASA7, SU2COR, TOMCATV and WAVE5. The others are integer benchmark programs - COMPRESS and ESPRESSO. We performed simulations on our 4 models by running the benchmark programs with different configurations of cache parameters such as cache size, line size, set associativity and so forth.

### 5.3.1 The NASA7 Benchmark

The NASA7 Benchmark is a collection of 7 kernel programs written in FORTRAN. The names and descriptions of the 7 kernels are shown in table 5.1.

| | |
|---|---|
| MXM | matrix multiply |
| CFFT2d | complex radix 2 FFT on 2D array |
| CHOLSKY | Cholesky decomposition in parallel on a set of input matrices |
| BTRIX | block tridiagonal matrix solution along one dimension of a four dimensional array |
| GMTRY | sets up arrays for a vortex method solution and performs Gausian elimination on the resulting arrays |
| EMIT | creates new vortices according to certain boundary conditions |
| VPENTA | inverts 3 matrix pentadiagonals in a highly parallel fashion |

Table 5.1: The NASA7 Benchmark

The programs are heavily floating point intensive and use double precision data. Most of the programs contain loops with intensive array references. Among the SPEC92 Benchmarks, NASA7 is the only one that contains arrays up to 4-dimension.

### 5.3.2 The SU2COR Benchmark

The SU2COR Benchmark is a vectorizable FORTRAN program with double precision floating-point arithmetic. In this application program from quantum physics, masses of elementary particles are computed in the framework of the Quark-Gluon theory. The data are computed with a monte carlo method taken over from statistical mechanics. The SU2COR Benchmark is a collection of 23 subroutines written in FORTRAN 77. They are MATMAT, INT2V, SWEEP, BESPOL and other subroutines.

### 5.3.3 The TOMCATV Benchmark

The TOMCATV Benchmark is a highly vectorizable double precision floating point FORTRAN benchmark. It is a vectorized mesh generation program and part of Prof. W.

Gentzsch's benchmark suite. It does little I/O and is described by Prof. Gentzsch as 90% - 98% vectorizable.

### 5.3.4 The WAVE5 Benchmark

The WAVE5 Benchmark is a large FORTRAN scientific benchmark with single precision floating point arithmetic. It is a 2-dimensional, relativistic, electromagnetic particle-in-cell simulation code used to study various plasma phenomena. WAVE5 is a modified version of WAVE benchmark. WAVE solves Maxwell's equations and particle equations of motion on a cartesian mesh with a variety of field and particle boundary conditions. The WAVE benchmark problem involves 500,000 particles on 50,000 grid points for 20 time steps.

The modification of WAVE5 from WAVE Benchmark leads to change in SPEC requirements:

- Benchmark was scaled down from 20 to 5 timestamps.
- Initialized some uninitialized ( assumed to be zero ) values.
- Disabled some code paths not used in the benchmark.
- Fixed some inconsistent common blocks.
- Normalized the code to prevent floating point exceptions in routine VSLV1p.
- Internal calculation of Elapsed time eliminated ( to permit SPEC mechanical verification ).

### 5.3.5 The COMPRESS Benchmark

The COMPRESS Benchmark is a CPU intensive integer benchmark with a significant I/O component. This benchmark contains no floating-point or array-valued calculations and is single-threaded. The COMPRESS benchmark is actually a compression application program. It reduces the size of the input file using adaptive Lempel-Ziv coding. The amount

of compression obtained depends on the size of the input, the number of bits per character, and the distribution of common substrings. In fact, compression and decompression programs are employed in a wide variety of applications which require storage and/or transmission of large text files. The vast majority of earlier SPEC benchmarks fells into the categories of engineering or scientific applications and performed trivial amounts of input/output. So, this benchmark can be of interest to some users who are not interested in the performance of CPU intensive engineering and scientific codes. This benchmark has already included a input text file in its code.

### 5.3.6 The ESPRESSO Benchmark

The ESPRESSO is an integer benchmark. It performs set operations such as union, intersect and difference. Sets are implemented as arrays of unsigned integers and set membership is indicated by a particular bit being ON or OFF. The set operations are implemented as sequences of logical AND's and OR's. The sets typically contain fewer than 200 members. The ESPRESSO Benchmark minimizes Boolean functions. It takes as input a Boolean function and produces a logically equivalent function possibly fewer terms. Both the input and output functions are represented as truth table. There are four input files *bca*, *cps*, *ti*, *tial* to the ESPRESSO. In our simulation, we used *ti* as the input file.

## 5.4 Simulations Parameters

The parameters of the simulation are as follows,

- The format of a read-in parameter file is as follows:

  <cache size, partition size, line size, set-associativity, start-up time, transfer time, best-expected cache performance, standard-expected cache performance>

  <program name, cycle execution time, cycle deadline period, type of program>

For Model 1 and Model 2, the cache size will be divided into half to create equal size of separate instruction cache and data cache. For Model 2 and Model 4, they will simply omit the partition size, best-expected and standard-expected cache performance. Type of program means that the program is either set to a periodic or aperiodic task.

- COMPRESS, ESPRESSO, NASA7, SU2COR and TOMCATV are set to be periodic tasks whereas Wave5 is set to be aperiodic task.
- Cache size takes the values of 8K, 16K and 32K bytes.
- Partition size takes the values of 0.5K, 1K and 2K bytes.
- Line size takes the values of 8 bytes, 16 bytes and 32 bytes
- Set associativity takes the values of 1-way, 2-way and 4-way.
- Best-expected cache performance ( hit ratio ) takes the values of 0.6, 0.7, 0.8, and 0.9.
- Standard-expected cache performance ( hit ratio ) takes the values of 0.5, 0.6, 0.7 and 0.8.
- Cycle execution time of each task takes the values of 1000 cycles, 5000 cycles and 10000 cycles.
- Cycle deadline period of each task takes the values of 6000 cycles, 30000 cycles and 60000 cycles.
- Start-up time takes the value of 6.
- Transfer time takes the value of 1.
- From all possible configurations of cache parameters used in our experiment, we defined a standard configuration. Each time we just vary one parameter of such

configuration to see the effects of changing that parameter. The standard configuration is shown on table 5.2.

| |
|---|
| Cache Size = 16k |
| Partition Size = 1k |
| Line Size = 16 bytes |
| Set Associativity = 2-ways |
| Best-expected Cache Performance = 0.9 |
| Standard-expected Cache Performance = 0.5 |
| Cycle Execution Time = 1000 cycles |
| Cycle Deadline Period = 6000 cycles |
| Start-up Time = 6 |
| Transfer Time = 1 |

Table 5.2: Standard Configuration Of Cache Parameters

Note that the start-up time and the transfer time will remain the same throughout the experiments.

# Chapter 6

# Analysis Of Simulations

## 6.1 Introduction

In this chapter, we will analyze the simulation results obtained from the experiments. First, we will show the statistics of the trace files we tested in the simulations. Then we will have an analytical explanations of the simulation results based on different values of cache size, partition size, line size, set associativity, best-expected and standard-expected cache performance. Thus, we not only can observe the performance differences of different models but also can observe the performance variations of the same model under different cache configurations.

## 6.2 Trace Files Statistics

The following table is the statistics of the six trace files which were run on our cache simulations. The statistics include the number of *LOAD* instructions, number of *STORE* instructions, number of *ALU* instructions, number of *BRANCH* instructions, number of *OTHERS* instructions and number of *DATA* executed.

| Trace File Name | Load | Store | ALU | Branch | Others | Data |
|---|---|---|---|---|---|---|
| NASA7 | 19258 | 6452 | 25608 | 32078 | 70586 | 25710 |
| ESPRESSO | 29717 | 12704 | 5244 | 29236 | 65941 | 42440 |
| SU2COR | 27084 | 27689 | 32279 | 23011 | 24544 | 54781 |
| COMPRESS | 8514 | 72838 | 1900 | 8664 | 24972 | 81360 |
| TOMCATV | 47309 | 20484 | 47539 | 10010 | 585 | 67803 |
| WAVE5 | 30264 | 9700 | 22052 | 22818 | 59646 | 40055 |

Table 6.1: The Statistics Of Trace Files

## 6.3 Interpretation Of Partial Cache Hit

In our simulations, we considered the partial cache hit as the case of cache miss. It is because we only foucs on the latency introduced by the memory system. Therefore, if the target instruction or datum in not in the cache but on the way of prefetching, we will first increment the cache miss counter by 1. Then we will add the remaining time cycles that the cache needs to wait for the coming instruction or datum. Such time cycles should be smaller than or equal to the cache miss penalty which is $C1 + C2 * ( N - 1)$ cycles. As a result, the figure which show the cache hit ratio and the cache miss ratio may not accurate. In fact, these figures are just shown for reference only.

## 6.4 The Effects Of Cache Size

In this simulation, we have examined 8K, 16K and 32K bytes cache size with our defined fixed configuration.

### 6.4.1 Performances Of Model 1, Model 2, Model 3 And Model 4

We should all know that when cache size increases, the cache hit ratio or performance should also increase. It is because large cache can hold more instruction and data. So, the cache misses ratio decrease. However, up to certain value of cache size, the cache

| |
|---|
| Cache Size = 8k or 16k or 32k |
| Partition Size = 1k |
| Line Size = 16 bytes |
| Set Associativity = 2-ways |
| Best-expected Cache Performance = 0.9 |
| Standard-expected Cache Performance = 0.5 |
| Cycle Execution Time = 1000 cycles |
| Cycle Deadline Period = 6000 cycles |
| Start-up Time = 6 |
| Transfer Time = 1 |

Table 6.2: Fixed Configuration With Different Cache Sizes

hit ratio will not increase any more and may even decrease. It is because the cache has become saturated now.

From the figure 6-1, we have observed that the above theory is true. When cache size increase from 8K to 32K bytes, the total task memory latency decrease. It is because large cache can store more instruction and data, which include those that has been stored in smaller cache. Therefore, the cache hit ratio will increase and hence the total memory latency can decrease. Also, we observe that the instruction and data hit ratio, and total tasks set utilizations are increased from 8K byte to 32K byte cache size.

First of all, we see that the degree of improvement of total instruction hit ratios is not as great as that of total data hit ratios. The reason is due to the sequential accessing pattern of instruction streams. Since we adopt one-block lookahead prefetching algorithm with always prefetch scheme. The highly sequential characteristic help to achieve and maintain the higher instruction hit ratio. However, since the access pattern of data streams is not in sequential feature, we can easily observe the great improvement in total data hit ratios because more data are now stored in the cache. As the cache hit ratios and total tasks memory latency improve when the cache size increases, the total tasks utilization also increase.

Second of all, when the cache size increase, the number of cache lines or entries increase so that the competition for the same entry decrease and hence the conflict misses decrease.

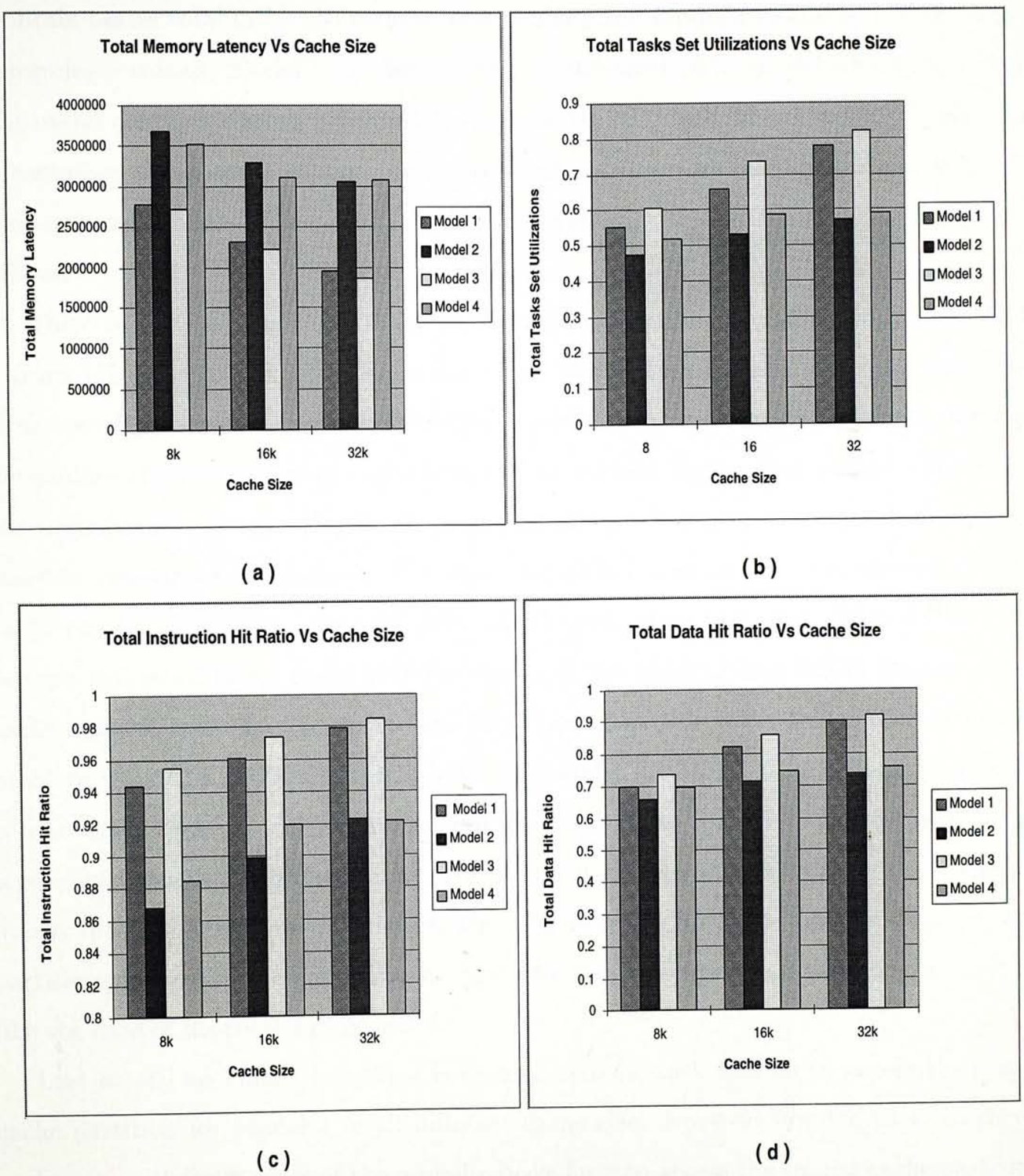


Figure 6.1: Models Performance Vs Cache Size

At the same time, the larger the cache, the more information 7can be stored and thus capacity misses decrease. As a result, the main reason to explain why Model 1 and Model 3 obtain better total tasks utilizations than Model 2 and Model 4 should be the problem of compulsory misses. Model 1 and Model 3 are partitioned cache model which can protect the cache contents during preemption whereas Model 2 and Model 4 not. Thus, after preemption the preempted task finds itself suffer from a serious compulsory misses in Model 2 and Model 4. Thus, the more the preemptions, the serious effect of compulsory misses.

Third of all, we clearly see that there is a little improvement of Model 2 and Model 4 from 16k byte to 32k byte cache size. But Model 1 and Model 3 do not have such situation. It is because under the real-time environment, the defined task execution time determines the percentage of cache that can be utilized during that period. The longer the period, the more the degree of cache utilization. However, the period is relatively short in real-time environment. For example, 1000 cycles in our case, so even if more cache can be allocated to the task, only a little portion will be used. That is, the cache become saturated faster under this environment. For Model 2 and Model 4, since all the cache memory will be allocated to the task, the cache will suffer this earlier saturation problem when the cache size is 32k byte. Especially for Model 4, it is because Model 2 is a separate cache model when will split the total cache space into halves to construct separate instruction and data cache. For Model 1 and Model 3, increasing the cache size means that the number of cache partitions will increase. Since the task uses one or more partitions instead of the entire cache, the cache saturation problem will not occur earlier like the case of Model 2 and Model 4.

Last of all, we found that there is no any periodic task needed to access the shared cache partition for Model 3 in all different cache size. However, for Model 1, when the cache size is 8k byte, some of the periodic tasks have to access the shared cache partition. But this phenomenon did not occur when the cache size are 16k and 32k byte. Model 1 is a separate cache model, the task will request the instruction cache partition to hold the instructions and the data cache partition to hold the data. When the cache size is 8k byte,

with 1k byte partition size, 4 instruction and 4 data cache partitions will be created. Since there are five periodic tasks, there are not enough instruction and data cache partitions to allow each task owning at least 1 instruction and 1 data cache partition. So, some of the periodic tasks must have to access the shared cache partition.

When the cache size is 16k byte, 8 instruction and 8 data cache partitions will be created. As one of them is needed to be the shared cache partition for the aperiodic task. Actually, we just have 7 instruction and 7 data partitions that can be allocated to the five periodic tasks. As a result, there will be a great chance for some periodic tasks to have just 1 instruction and 1 data cache partitions. From the rule of the dynamic cache partition re-allocation, if a shared cache partition created, when the cache hit ratio of any periodic task below the standard-expected cache performance value, it need to give out its partition(s) and then access to the shared partition. However, those tasks which own 1 instruction and 1 data cache partition did not access the shared cache partition. That is even if the task just owns one instruction or data cache partition to hold the instructions or data, the cache hit ratio still above the standard-expected cache performance value. In fact, such situation is similar to the case when the cache size is 8k byte for Model 3.

In this simulation, the standard-expected performance value is 0.5 or 50% Therefore, we now know an important fact that for 1k byte cache partition size, the task can obtain a cache hit ratio no less than 0.5 or 50%. In fact, this result is similar to what Kirk[Kir88], Hennessy and Patterson[HP90] mentioned. They stated that cache as small as 1k byte or 256 word can achieve hit ratio above 70%. As Model 3 is an unified cache model, even when the cache size is 8k byte, there will be 8 cache partitions created which is similar to the case of 16k byte cache size for Model 1.

## 6.5 The Effects Of Cache Partition Size

In this simulation, we have examined 0.5K, 1K and 2K bytes cache partition size with our defined fixed configuration.

Such simulations are concerned with Model 1 and Model 3. There is no any observable

| |
|---|
| Cache Size = 16K |
| Partition Size = 0.5k or 1k or 2k |
| Line Size = 16 bytes |
| Set Associativity = 2-ways |
| Best-expected Cache Performance = 0.9 |
| Standard-expected Cache Performance = 0.5 |
| Cycle Execution Time = 1000 cycles |
| Cycle Deadline Period = 6000 cycles |
| Start-up Time = 6 |
| Transfer Time = 1 |

Table 6.3: Fixed Configuration With Different Cache Partition Sizes

effect on Model 2 and Model 4 because these models utilize the whole ( non-partitioned ) caches rather than partitioned caches.

For partitioned cache model, the whole cache is divided into many small cache partitions and then allocate to each task. The extra cache partition will be allocated to the task, if possible, after it requested. Therefore, the cache space that the task accessed is built up from the cache partition one by one. In Chapter 3, we have stated that there should be a sharper rise in the cache performance for partitioned cache model with increasing partition size. It is because when the cache partition size increase, the larger cache partition can hold more instruction and data when compared with smaller cache partition. Also, the number of set in the cache partition increase. Thus, the capacity and conflict misses will decrease, so the cache hit ratios will increase. This situation is somehow similar to the case of increasing the total cache size.

In our simulation, we have fixed the total cache size and then varied the partition size. Thus, it means that the number of cache partitions will decrease with the increase in cache partition size. The total cache size used in such simulation is 16k byte. So, when the partition size are 0.5k, 1k and 2k byte, then the total number of cache partitions are 32, 16 and 8 respectively. For separate cache these number of cache partitions needed to be further divided by half. One of them will be used as a shared cache partition for an aperiodic task to access.

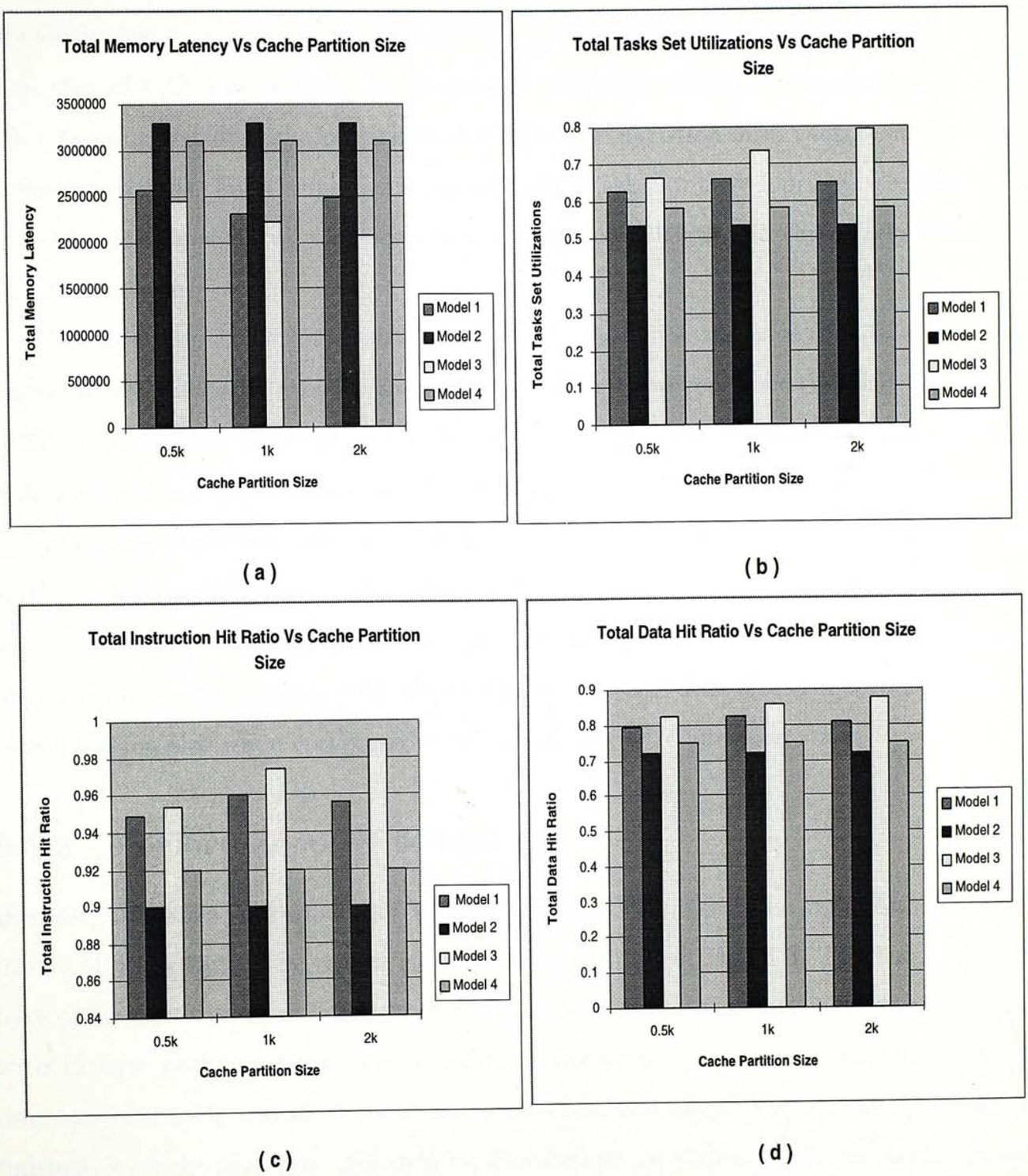


Figure 6.2: Models Performance Vs Cache Partition Size

### 6.5.1 Performance Of Model 3

At the beginning of the experiments, the periodic tasks will request the cache partition. Obviously, the cache partition which have 2k byte large will have a higher cache hit ratio than that of 0.5k and 1k byte. In the case of Model 3, which is an unified cache model. After running out of cache partitions, for 0.5k byte partition size, each task can have 6 private partitions. For 1k byte partition size, each task can have 3 private partitions. For 2k byte partition size, two tasks can have 2 private partitions and three tasks can have 1 private partition.

Therefore, the cache hit ratio of the periodic tasks for the case of 0.5k and 1k byte partition size will catch up with the case of 2k byte. However, since the tasks for the case of 2k byte partition size does not suffer form the higher cache misses initially, the same task must execute faster than that of other cases especially for those tasks which have two 2k byte cache partitions. Assume that there is one task finished the execution, its cache partitions will be allocated to the others tasks. No doubt, the 2k byte cache partition can help a lot to the task when compared to the 0.5k and 1k byte cache partition based on the cache miss rate decrease with the cache size increase. It is also true for the case of 1k byte partition size when compared with case of 0.5k byte partition size.

### 6.5.2 Performance Of Model 1

Basically the cache performance of Model 1 improves when the cache partition size increase from 0.5k to 1k byte. The reason is the same as the case of Model 3. The tasks with 0.5k byte cache partition must suffer from a higher conflict and capacity misses than the task with 1k byte cache partition initially. After running out the cache partitions, the tasks own approximately the same cache space for these two cases. Once a periodic task has finished, its cache partitions will then be allocated to someone else. So, the cache hit ratio will improve much more when allocating a bigger partition. Thus, the cache performance increase from 0.5k to 1k byte partition size.

But Model 1 is a separate cache model, when the partition size is 2k byte, there will

be 8 partition totally for separate instruction and data cache with 16k byte total cache size. As 1 partition must be used as a shared cache partition for aperiodic task to execute, there are only 3 partition left for the five periodic tasks. As a result, three periodic tasks will own 1 private cache partition and the others will access the shared partition. In our rule, at most two periodic tasks needed to access the shared partition. Although the three tasks which owns 2k cache partition will have a higher cache hit ratio than the same tasks in the case of 0.5k and 1k byte partition size. However, as two periodic tasks access the shared cache partition, they will have poor cache hit ratio. Fortunately, a large partition size will have lower conflict and capacity misses. The decrease in these misses help to compensate the compulsory misses during preemption. In addition, when the periodic task which owns private cache partition has finished, its partition can be assigned to the task which access the shared partition. As a result, the performance lost will not be too large. Although the cache performance is poor than the case of 1k byte, it still better than the case of 0.5k. It is because the cache hit ratio of aperiodic task plays a great role to the total cache hit ratio. When the cache partition size is 0.5k byte, the aperiodic task suffers the lowest cache hit rate whereas it suffers the highest cache hit rate when the partition size is 2k byte. Finally, from the figure 6-2, we see that the cache performance increase from 0.5k to 1k and then degrade from 2k byte partition size.

## 6.6 The Effects Of Line Size

In this simulation, we have examined 8, 16, and 32 bytes line size with our defined fixed configuration.

### 6.6.1 Performance Of Model 1, Model 2, Model 3 And Model 4

Each time when a line is fetched from main memory, it takes $C1 + C2 * (N - 1)$ cycles. The larger the line size, the larger the value of $N$. So, the transfer time for a cache line does

| |
|---|
| Cache Size = 16k |
| Partition Size = 1k |
| Line Size = 8 or 16 or 32 bytes |
| Set Associativity = 2-ways |
| Best-expected Cache Performance = 0.9 |
| Standard-expected Cache Performance = 0.5 |
| Cycle Execution Time = 1000 cycles |
| Cycle Deadline Period = 6000 cycles |
| Start-up Time = 6 |
| Transfer Time = 1 |

Table 6.4: Fixed Configuration With Different Line Sizes

increase with line size. However, it also means that the blocks in each cache line increases as the line size increases. Hennessy and Patterson[HP90] stated that increasing line size lowers the compulsory miss rate until the reduced misses of larger blocks ( spatial locality ) are outweighed by the increased conflict misses as the number of entry per set shrinks ( temporal locality ). As a result, when the line size increase, the average memory-access time decrease initially and then become constant. Finally, the average access time will increase again [HP90]. Thus, from the figure 6-3, we see that the total tasks memory latency decrease with the increase in the line size since the average access time decrease.

Generally, the cache performances of four models improve as the line size increases. It is because increasing line size means that more requested instruction or data are in the cache. As said before, instruction stream access is more likely in sequential pattern, by using one block lookahead prefetching algorithm and read-only prefetch buffer, the effects of increasing line size on instruction hit rate is not as obviously as data hit rate. Although the data access pattern is not likely in sequentially, when increasing the line size and using prefetching, it will increase the probability of getting the next requested data before it is actually referenced.

For non-partitioned cache models which need to suffer from the compulsory misses each time when the preempted task resumes its execution whereas partitioned cache model does not. Increasing the line size should make more contribution to non-partitioned cache

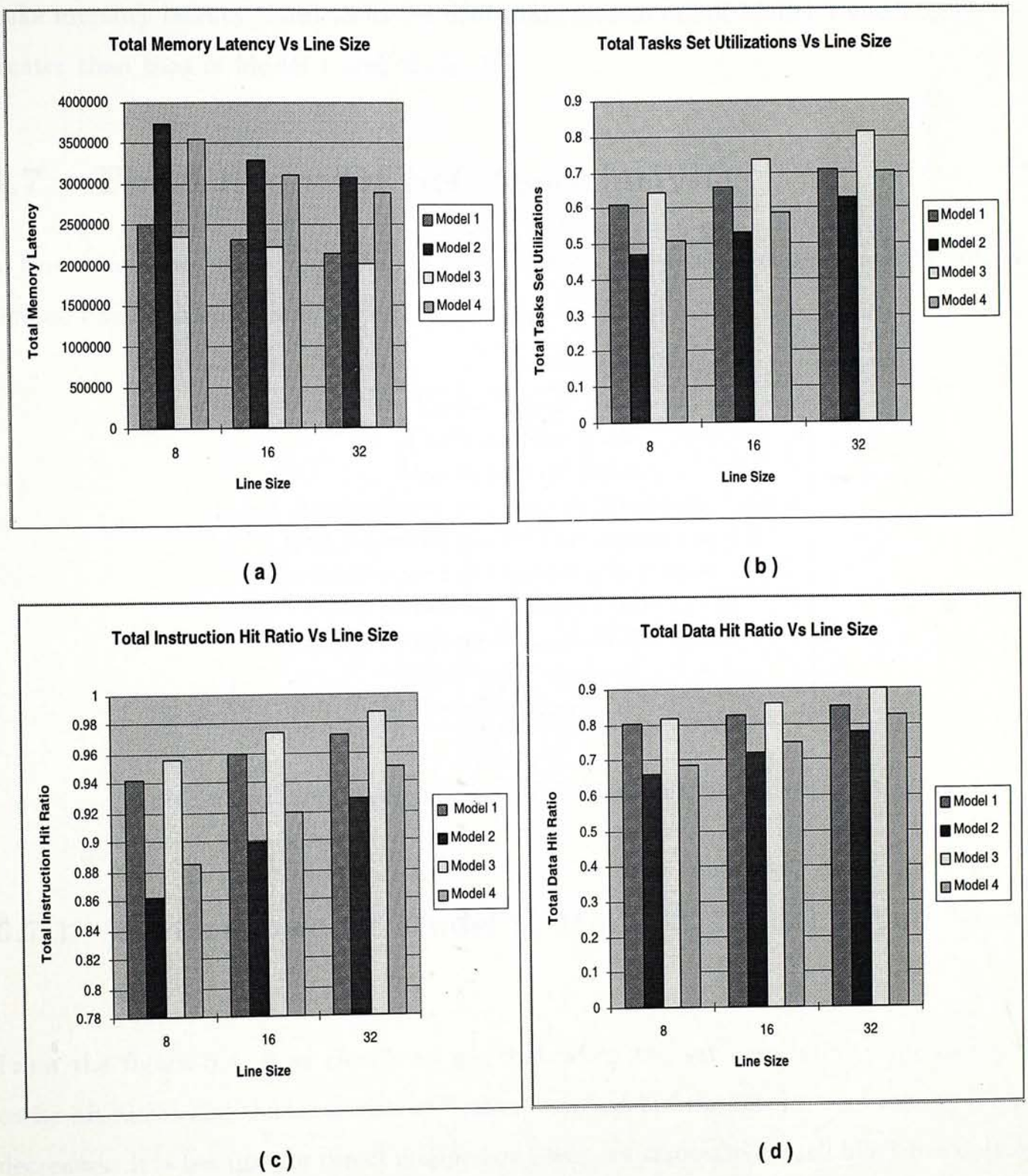


Figure 6.3: Models Performance Vs Line Size

models. It is because the longer the line size, the lower the compulsory misses which was mentioned above. As a result, the cache performances improvement, such as the total tasks memory latency, total tasks set utilization and so on, of Model 2 and Model 4 are greater than that of Model 1 and Model 3.

## 6.7 The Effects Of Set Associativity

In this simulation, we have examined 1-way, 2-ways and 4-ways set associativity with our defined fixed configuration.

| |
|---|
| Cache Size = 16k |
| Partition Size = 1k |
| Line Size = 16 bytes |
| Set Associativity = 1-way or 2-ways or 4-ways |
| Best-expected Cache Performance = 0.9 |
| Standard-expected Cache Performance = 0.5 |
| Cycle Execution Time = 1000 cycles |
| Cycle Deadline Period = 6000 cycles |
| Start-up Time = 6 |
| Transfer Time = 1 |

Table 6.5: Fixed Configuration With Different Set Associativities

### 6.7.1 Performance Of Model 1, Model 2, Model 3 And Model 4

From the figure 6-4, it is clearly to see that when the set associativity increases, the cache hit ratios and the total task utilization increase but the total tasks memory latency decreases. It is because for direct mapped or 1-way set associative, each block has only one place that can map to the cache. In fact, 1-way set associative always lead to the highest number of competitions for the same entry and thus obtain the highest conflict misses. As a result, the lower the set associative, the higher the cache misses rate. Therefore, as

the set associative increases from 1-way to 2 ways and then to 4-ways, the cache hit ratio increases. It is because the conflict misses decrease since the chance of competition to the same places decrease. When the cache hit ratio increases, the total tasks memory latency should decrease. When the cache hit ratio increases and the memory latency decreases, the total task set utilization will increase.

In addition, we have observed that the partitioned cache models, Model 1 and Model 3, obtained better results than that of non-partitioned cache models, Model 2 and Model 4. The main difference between these two models is that one can protect the cache content during preemption whereas the other can not. Although the increase in set associativity will lead to the decrease in conflict misses, non-partitioned cache models must suffer from compulsory misses again and again after preemption. In contrast, partitioned cache models do not have compulsory misses problem after preemption. As a result, the performance of partitioned cache models are better than that of non-partitioned cache models.

## 6.8 The Effects Of The Best-expected Cache Performance

In this simulation, the values of the best-expected cache performance that we have examined are 0.6, 0.7, 0.8 and 0.9 with our defined fixed configuration.

Such simulation are mainly concerned with Model1 and Model3. It is because the best-expected cache performance value is used in the dynamic cache partition re-allocation.

In the dynamic cache partition re-allocation, the best-expected cache performance value is used to inform the periodic task how many cache partitions it should owns. When the value becomes large, the periodic task will try to request and hold more cache partitions. During the dynamic cache partition re-allocation, the system will first check if all periodic tasks obtain the best-expected performance value. If yes, then it exits the partition re-allocation procedure and not otherwise. So, when the value is small and

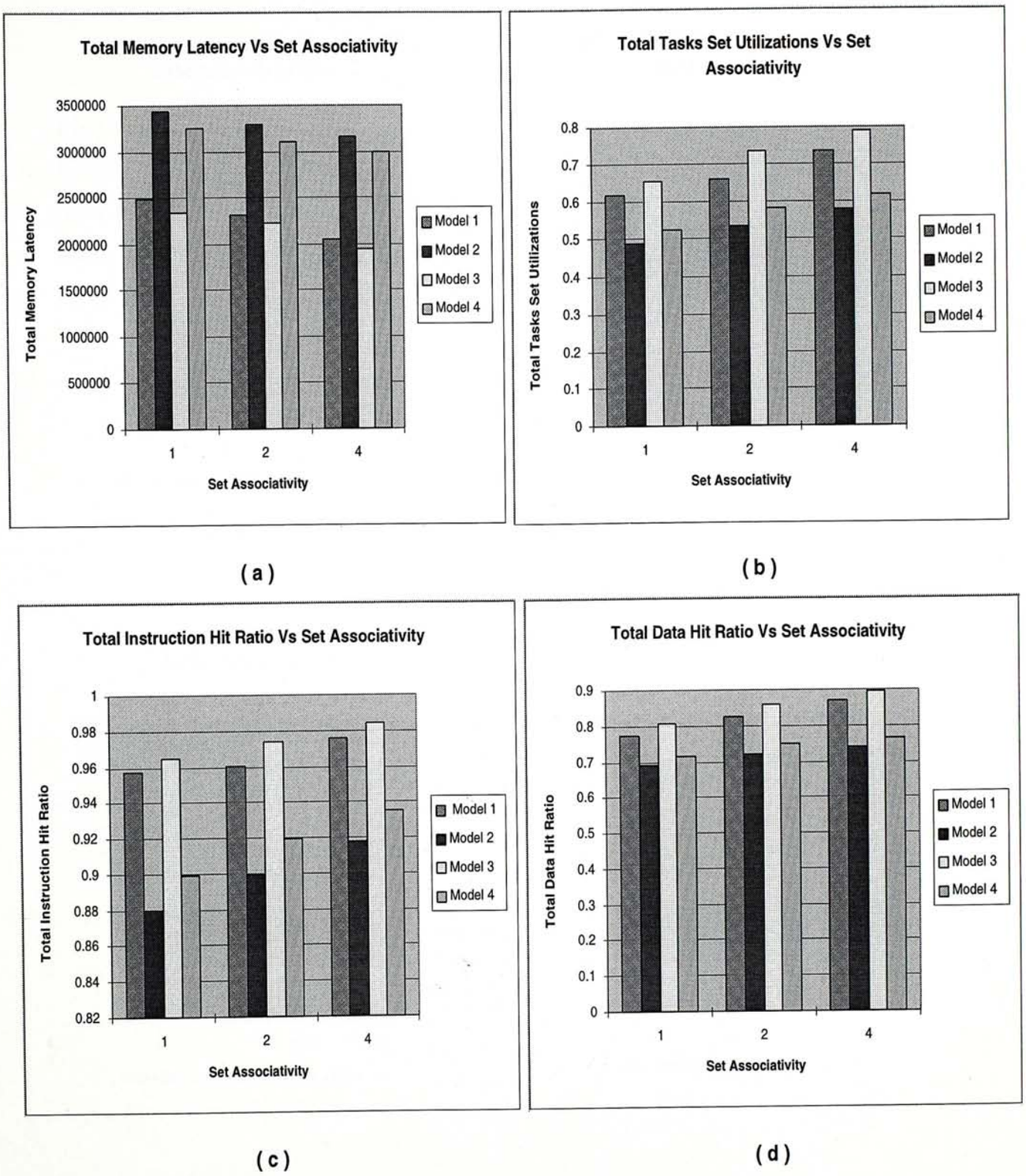


Figure 6.4: Models Performance Vs Set Associativity

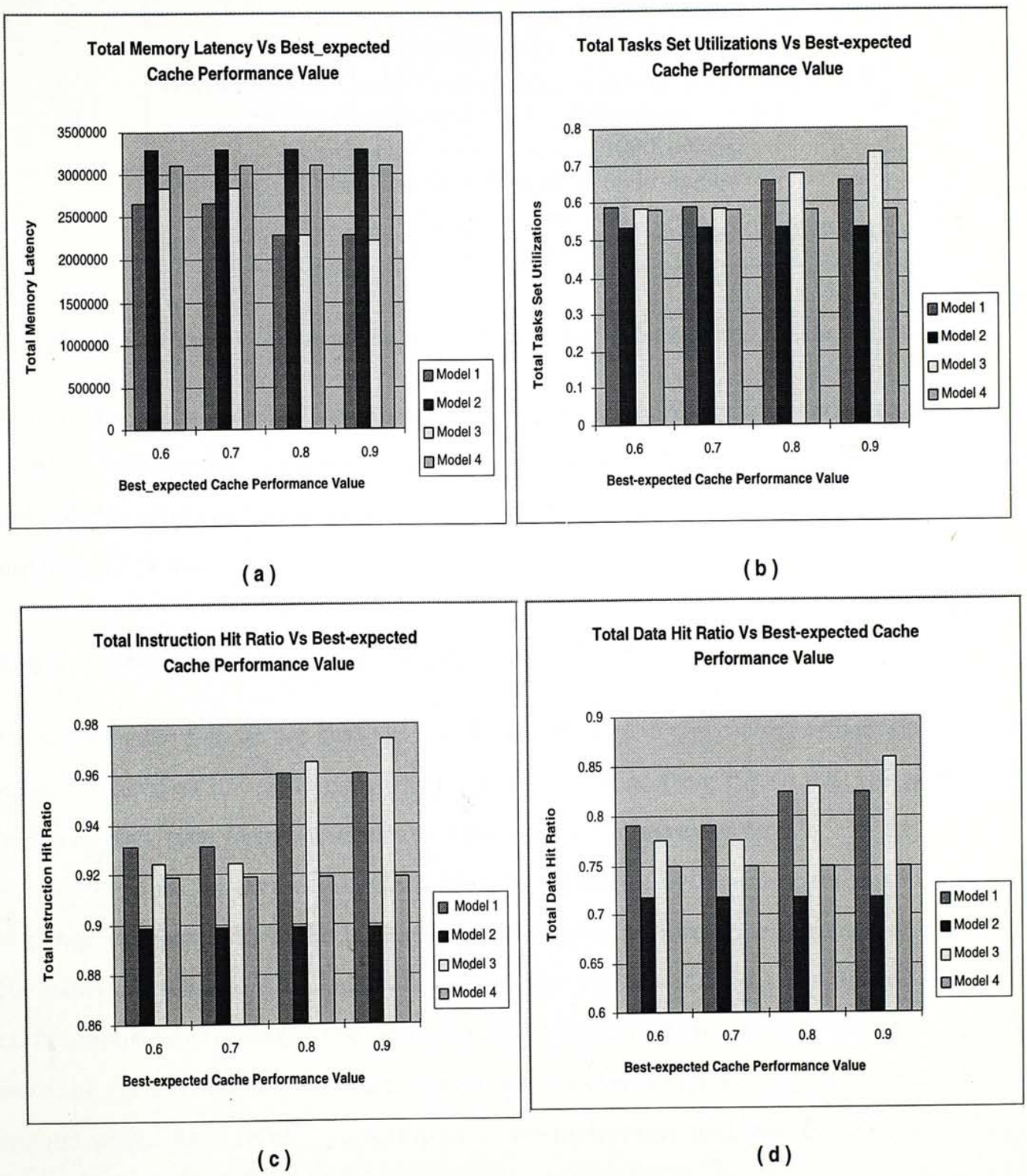


Figure 6.5: Models Performance Vs Best-expected Cache Performance Value

| |
|---|
| Cache Size = 16k |
| Partition Size = 1k |
| Line Size = 16 bytes |
| Set Associativity = 2-ways |
| Best-expected Cache Performance = 0.6 or 0.7 or 0.8 or 0.9 |
| Standard-expected Cache Performance = 0.5 |
| Cycle Execution Time = 1000 cycles |
| Cycle Deadline Period = 6000 cycles |
| Start-up Time = 6 |
| Transfer Time = 1 |

Table 6.6: Fixed Configuration With Different Best-expected Cache Performances

all periodic tasks have achieved such value, even if there are free cache partitions, the partition will not be allocated to any tasks. For Model 1, there are totally 8 instruction and 8 data cache partitions whereas there are 16 cache partitions for Model 3. Each cache partition is 1k byte large.

### 6.8.1 Performance of Model 1

From the figure 6-5, we see that there is no any change in the cache performance when the value are 0.6 and 0.7. However, when the value change from 0.7 to 0.8, the performance increases and then becomes constant when the value increases to 0.9.

It is because, in the section 6.3.1, we stated that 1k byte cache partition can enhance the task at least 70% cache hit ratio, so the cache performance remain the same when the value are 0.6 and 0.7. Thus, all five periodic tasks just hold 1 instruction and 1 data cache partition can obtain the best-expected value which are 0.6 and 0.7. In both cases, once the system enters the dynamic cache partition re-allocation procedure, they will exit immediately. As a result, we found that two instruction and two data cache partitions did not use through out the experiment.

When the value increases to 0.8, the task needs extra cache partition to achieve. When the task holds more cache partitions, the cache performance must increase definitely. At that time, all instruction and data cache partitions will be used. Since all periodic tasks

now must obtain the cache hit ratio higher than the standard-expected cache performance value, no any victim task will be chosen to give out its cache partitions. Thus, even when the value increases to 0.9, the cache performance will not increase because of no free cache partition that can be used.

### 6.8.2 Performance of Model 3

From the figure 6-5, we see that there is no any change in the cache performance when the value are 0.6 and 0.7. After that, when the value increases, the cache performance increases.

It is because, in the section 6.3.1, we stated that 1k byte cache partition can enhance the task at least 70% cache hit ratio, so the cache performance remain the same when the value are 0.6 and 0.7. Thus, all five periodic tasks just hold 1 cache partition can obtain the best-expected value which are 0.6 and 0.7. In both cases, once the system enters the dynamic cache partition re-allocation procedure, they will exit immediately. As a result, we found that ten cache partitions did not use through out the experiment.

When the value increases to 0.8, the task needs extra cache partition to achieve. As ten cache partitions are free to allocate, then each task will be allocated one more cache partition. In the dynamic cache partition re-allocation, when there are free cache partitions, they will allocate to the periodic tasks based on their cache hit ratio. That is, totally each task will own two cache partitions. When the task holds more cache partitions, the cache performance must increase definitely. Actually, all periodic tasks can obtain the cache hit ratio above 80%. So, ten cache partitions are used but five cache partitions are not used when the best-expected value is 0.8.

When the value increase to 0.9, the remain five free cache partitions will be used by all the periodic tasks. Since the larger the cache space, the higher the cache hit ratio and thus the cache performance increase.

## 6.9 The Effects Of The Standard-expected Cache Performance

In this simulation, the values of the standard-expected cache performance that we have examined are 0.5, 0.6, 0.7, and 0.8 with our defined fixed configuration.

| |
|---|
| Cache Size = 16k |
| Partition Size = 1k |
| Line Size = 16 bytes |
| Set Associativity = 2-ways |
| Best-expected Cache Performance = 0.9 |
| Standard-expected Cache Performance = 0.5 or 0.6 or 0.7 or 0.8 |
| Cycle Execution Time = 1000 cycles |
| Cycle Deadline Period = 6000 cycles |
| Start-up Time = 6 |
| Transfer Time = 1 |

Table 6.7: Fixed Configuration With Different Standard-expected Cache Performances

Such simulations are mainly concerned with Model1 and Model3. It is because the standard-expected cache performance is a reference point used in the dynamic cache partition re-allocation.

During the dynamic cache partition re-allocation, the standard-expected cache performance value is used to find the victim periodic tasks to give out their cache partitions. After that, the victim task needs to access the shared cache partition. Thus, the larger the value, the more chance for a task to give out the partition. For Model 1, there are totally 8 instruction and 8 data cache partitions whereas there are 16 cache partitions for Model 3. Each cache partition is 1k byte large.

### 6.9.1 Performance Of Model 1

From the figure 6-6, we see that when the value are 0.5, 0.6 and 0.7, the cache performances of Model 1 remain no change. That is, even though the task owns 1 instruction and 1 data cache partition, the cache hit ratio can still reach at least 70%.

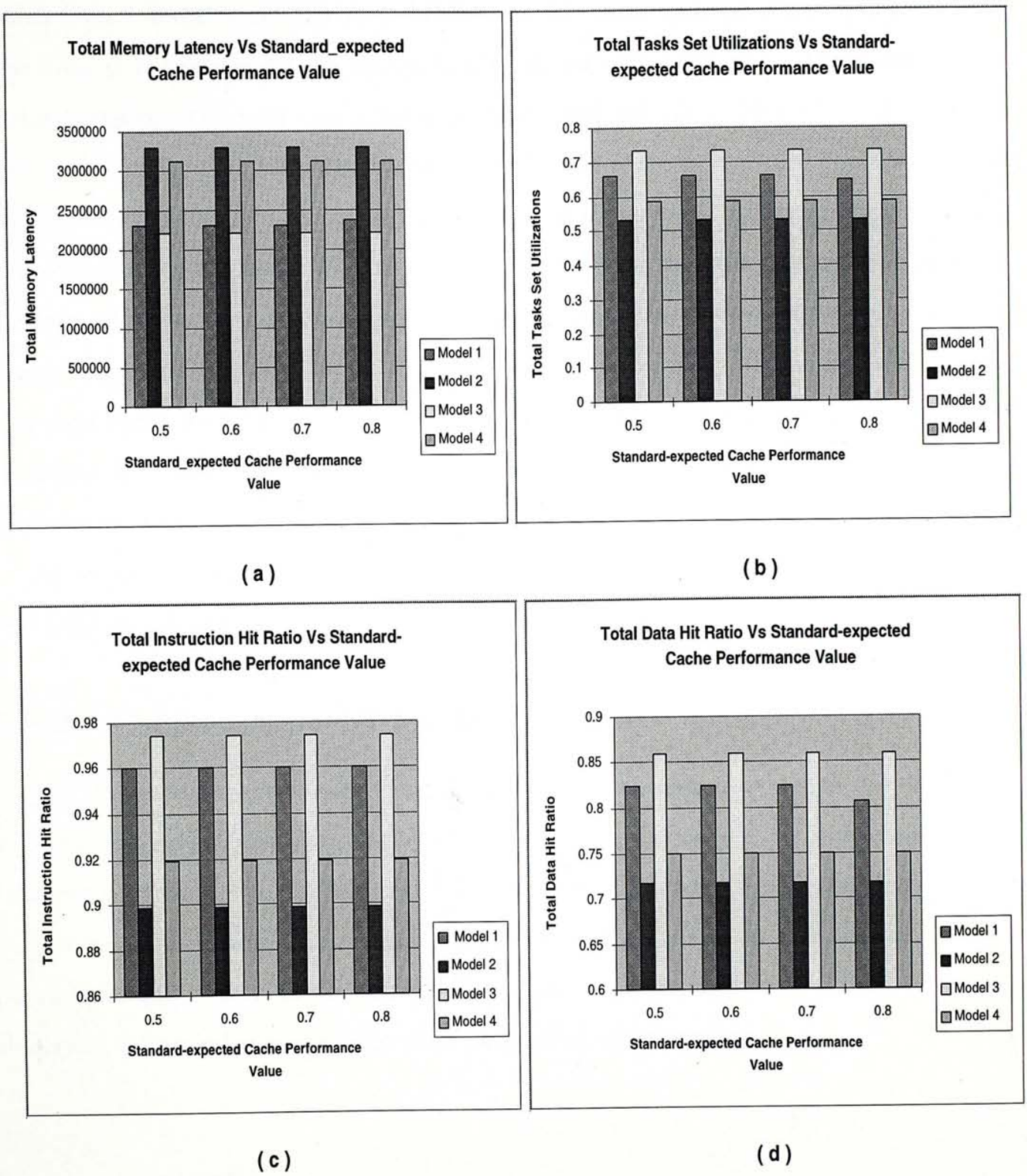


Figure 6.6: Models Performance Vs Standard-expected Cache Performance Value

When the value increase to 0.8, the data hit rate degrades but the instruction hit rate still remain unchanged. It is because at that moment at most two periodic tasks which own 1 data cache can not obtain data hit ratio at least 80%. So, they have to free their partitions and then access to the shared cache partition. With only 1 private cache partition, the task can achieve at least 70% hit rate. But when they access to the shared cache partition, the cache hit ratio will decrease because the cache contents will be changed during the preemption by preemptiong task. Although there will be the improvement for the tasks which gain these partitions, the gain can not compensate for the lost. After a periodic task finished the execution, its owned cache partition will be free and thus can allocate to the others. However, if the task still can not obtain at least 80% data cache hit rate, that task needs to give out the partition and access to the shared partition. So, there is a frustration of the data cache performance and hence lead to the decrease in the data cache hit ratio.

As we clearly see that the total instruction hit ratio is above 80%, so there is no any performance degrade in the instruction hit ratio.

### 6.9.2 Performance Of Model 3

For all standard-expected cache performance value we tested, which is 0.5, 0.6, 0.7 and 0.8, the instruction and data cache hit rates remain unchanged. It is because there are 15 cache partitions that can allocate to the five periodic tasks. On average, each task can have 3 cache partitions. Since 1 cache partition can guarantee at least 70% hit rate, it is possible for those tasks to achieve 80% hit ratio. As a result, there is no any performance change in all tested standard-expected cache performance value.

## 6.10 The Effects Of Cycle Execution Time/Cycle Deadline Period

In this simulation, we have examined 1000/6000, 5000/30000 and 10000/60000 cycles of cycle execution time and cycle deadline period respectively with our defined fixed configuration.

| |
|---|
| Cache Size = 16K |
| Partition Size = 1K |
| Line Size = 16 bytes |
| Set Associativity = 2-ways |
| Best-expected Cache Performance = 0.9 |
| Standard-expected Cache Performance = 0.5 |
| Cycle Execution Time = 1000 or 5000 or 10000 cycles |
| Cycle Deadline Period = 6000 or 30000 or 60000 cycles |
| Start-up Time = 6 |
| Transfer Time = 1 |

Table 6.8: Fixed Configuration With Different Cycle Execution Time And Cycle Deadline Period

### 6.10.1 Performances Of Model 1, Model 2, Model 3 And Model 4

Remind that the main differences between the general-purpose computing environment and the real-time computing environment are the number of preemptions and the length of task cycle execution time. In real-time computing environment, the task cycle execution time is much shorter than that in general-purpose computing environment. Therefore, the number of task preemptions become more frequently. During the preemption, the cache contents will be flushed out and the preempted task will suffer from compulsory misses. So, the more the preemptions, the serious the compulsory misses and vice versa.

When the task cycle execution time increases, it means that the task can have more time to utilize the cache. So, a large cache should be supplied because it can help to

decrease the conflict and capacity misses. Thus, the cache model which can provide large cache space for the task to execute would be much better than that of small cache space. In the figure 6-7, it shows that Model 2 and Model 4 gave a better simulation results than Model 1 and Model 3 when the cycle execution time increases. In fact, Model 2 and Model 4 still suffer from compulsory misses during preemption. But, as the number of preemptions decrease, the compulsory misses become less important. Since more time is allowed for the task to execute, there will be less chance for a task to be swapped out during the cold start period of the cache. After the cold start period, the cache will contain some of the frequently used instruction and data lines, so the compulsory misses will decrease and thus cache hit ratios will increase. As said before, compulsory misses are the main factor which lead to the unpredictable performance of non-partitioned cache models. If the compulsory misses can be reduced by decreasing the number of preemption, then the performance of non-partitioned cache models would increase undoubtedly.

As Model 1 and Model 3 use one or more cache partitions, the smaller cache space will suffer from a higher capacity misses and conflict misses. Therefore, their performances will be degraded when the cycle execution time increases. Actually, these problems can be solved by using dynamic cache partitions re-allocation. In our re-allocation scheme, each time a task can request only one cache partition in each dynamic cache partitions re-allocation. And the cache partitions re-allocation will only perform during each "round" of tasks execution. However, the number of rounds are inversely proportional to the cycle execution time. So, the longer the cycle execution time, the less number of the cache partitions re-allocation. As a result, when compared with the non-partitioned cache models, the longer cycle execution time will cause the partitioned cache models which provide smaller cache space to the task a higher capacity misses and conflict misses. Eventually, the cache performances of Model 1 and Model 3 degrade as the cycle execution time increase. That is, the total task memory latency, the instruction and data cache hit ratio, and the total task set utilization will decrease.

Thus, when the task cycle execution time increases, the cache performances of Model 2 and Model 4 will increase whereas Model 1 and Model 3 will decrease.

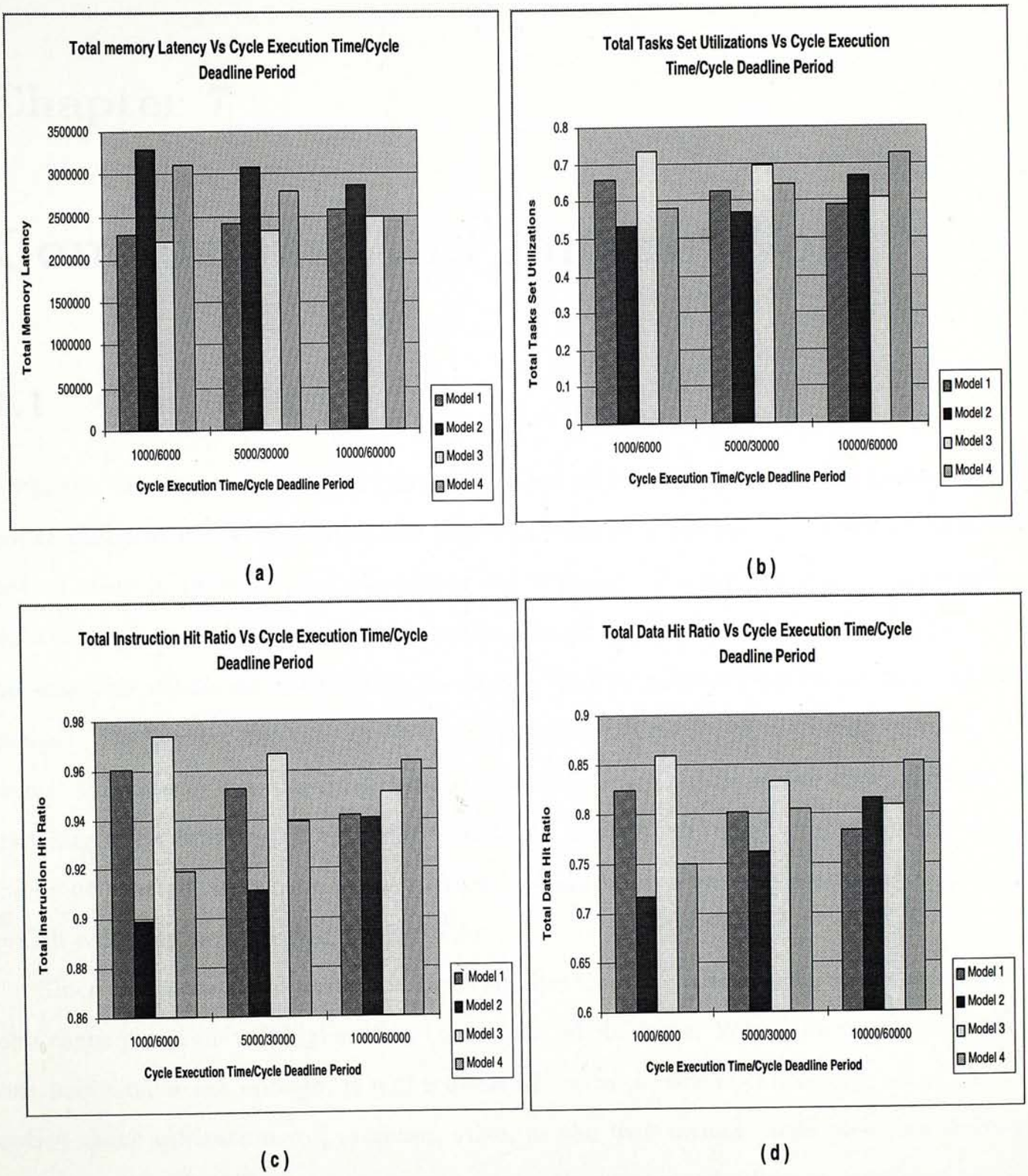


Figure 6.7: Models Performance Vs Cycle Execution Time/Cycle Deadline Period

# Chapter 7

# Conclusions And Future Work

## 7.1 Conclusions

From the simulation results, we can justify what we have said at the very beginning. The entire cache memory which does not protect the cache contents during the preemption is not suitable in the real-time computing environment. Even if the entire cache memory does protect the cache contents, the utilization of the cache memory is not higher. It is because the utilization of the cache memory is directly proportional to the task execution period. The shorter the task to execute, the little utilization of the cache memory. This is just the case in the real-time computing environment. Most of the time, the task just executes a few hundred to thousand cycles. As a result, to remedy the situation of lower cache utilization, the entire cache memory should be divided into many small partitions which will then be allocated to the tasks.

Since the cache partition size is much smaller than the whole cache size, the utilization of a cache partition will higher than that of the whole cache. When the task finds out that one partition is not enough, it will request an extra partition. Therefore, as a whole the entire cache utilization will increase. Also, as the task owned cache partition is private to that task only, the cache performance can be guaranteed when compared with using the whole cache which is subject to be over-written by the preempting task during the preemption. Actually, we have shown that Model 1 and Model 3 which are partitioned cache models have obtained a higher tasks set utilizations than non-partitioned cache

models. Most of the time, the tasks utilizations even greater than the bound suggested by Liu and Layland[LL73] in 1973. They claimed that the bound will be 0.69 or $ln\ 2$. Also, we discovered that for 1k byte cache partition size, the task can obtain a cache hit ratio no less than 0.7 or 70%.

In the proposed cache models, the number of cache partitions that a task owned is not restricted in the power of 2. If the number is restricted in the power of 2, one or some cache partitions that a task owned may not be utilized at all and thus become wasted. It is because each time when the task successfully requests cache partitions, the number of its owned cache partitions will be doubled. Therefore, the same problem of above will occur. Hence, the best way to do is not to restrict the number of cache partitions in the power of 2, and also only one cache partition is allocated to the task each time. As a matter of fact, each cache partition can be fully utilized.

When the cache size of all models increase, the performance of all cache models will increase. It is because cache misses rate decrease as cache size increase. The larger the cache, the more number of cache lines and thus the more the information can be stored. As a result, capacity and conflict misses will decrease. Also, the increase in total cache size should be more important for partitioned cache models than non-partitioned ones. It is because under the real-time computing environment, the cache memory will become saturated much earlier than under general-purpose computing environment. Thus, a large portion of the entire cache memory will not be utilized at all. Also, the task must suffer from the compulsory misses every time once it resumes its execution. In contrast, for partitioned cache models, when the total cache size increase, it means that the number of cache partitions that can be allocated to the tasks will increase. As the tasks in such cache models utilize one or some of the cache partitions. Therefore, the cache utilization can be increased. In addition, the cache contents of the private cache partition will not be changed during the preemption, so the overall effect by decreasing both the capacity and conflict misses, and seldom compulsory misses make the cache performance of partitioned cache models superior than non-partitioned ones.

Generally, increasing the cache partition size can increase the cache performance for

partitioned cache models. It is because the size of a large cache partition may double or more than that of a small cache partition. Since the tasks in partitioned cache models request one cache partition in each allocation, larger cache partition will help to decrease both the conflict and capacity misses than small cache partition. However, with the fixed total cache size, the cache partition size can not be too large. It is because the number of cache partitions decrease with the increase in partition size. The best is that all periodic tasks have their own cache partitions. If the number of cache partitions smaller than the number of periodic tasks, some of the tasks must have to access the shared cache partition and so obtain a poor cache hit ratio. As a result, the total cache performance will degrade.

Increasing the block size can improve the cache performance because when the block size increases, more information or a larger extent of spatial locality of the reference can be stored in the cache. However, larger block size will increase the transfer time of the information and also may bring too many useless information into the cache, so the block size can not be too large.

We have observed that increasing set associativity can help to increase the performance of all models. It is because the number of collisions for competing the same entry decrease as set associativity increase. Hence, the cache hit ratios will increase. Direct-mapped or 1-ways set associative is the worst scheme as each instruction or data block must be mapped to the only one place in the cache. In contrast, fully associative is the best as each instruction or data block can go to any place in the cache. In fact, we should notice that the time for searching the cache increases directly proportional to the set associative. Therefore, there will be a tradeoff between the cache performance and the set associative. Usually, most systems implement no more than 8-ways set associative cache memory in order to increase the cache performance. At the same time, the time for searching the cache will not be too long.

During the simulation, we found that when the best-expected cache performance value increases, the cache performance of partitioned cache models will increase. It is because if the value is set to a small value, there will be a great chance for all tasks to achieve it.

So, once the system enters the dynamic cache partition re-allocation procedure and finds out that all periodic tasks have obtain such value, it will exit immediately. That is, even if there are free cache partitions, these partitions will not be allocated to the tasks. As a result, the cache performance will not increase because of the limited cache size of each task owned. Thus, when the best-expected value becomes large, in order to achieve such goal, the periodic tasks will request and compete the extra cache partition. Eventually, the larger the cache space, the higher the cache hit ratio. However, it also depends on the number of cache partitions that can be allocated to the tasks. Therefore, when there is not enough cache partitions, even if we set the best-expected performance value as a large value, the cache performance can not improve so much.

Besides the best-expected cache performance value, the other performance value that we used in the dynamic cache partition re-allocation is the standard-expected cache partition value. This value is used to find out the victim periodic task, which need to give out its owned cache partition, when there is a shortage of cache partition. Basically, when there are enough cache partitions for the periodic tasks, usually they can obtain a higher cache hit ratios. So, even if the value is set to large, there will be no any effect to the cache performance. However, if there are not enough cache partitions for the periodic tasks which lead to the cache hit ratios of some tasks below the standard-expected performance value, then these tasks will give out their partition and then access the shared cache partition. Since the task which access the shared cache partition will suffer from a poor cache hit ratio, the cache performance will degrade even though there will be a performance gain from the task which obtain the extra partition.

As said in Chapter 3, the more the preemptions, the less the predictable of the non-partitioned cache models. Actually, the number of preemptions depend on the cycle execution time. The shorter the cycle execution time, the more the preemption. Therefore, when the cycle execution time and cycle deadline period are 1000 and 6000 cycles. The non-partitioned cache models suffer from the highest cache misses rate and thus become highly unpredictable. It is because the number of the compulsory misses are the highest. When the cycle execution time and cycle deadline period increase, each task can have

more time to establish the cache contents. Remember that for non-partitioned cache models, each task still suffers from the compulsory misses when it resumes its execution after preemption. However, as more time is allowed for the task to execute, there will be less chance for a task to be swapped out during the cold start period of the cache. After the cold start period, the cache will contain some of the instruction lines and data lines, so the compulsory misses will decrease and thus cache hit ratios will increase. As a result, the performance of non-partitioned cache models will improve when the cycle execution time and cycle deadline period increase. However, for the partitioned cache models, since the tasks use one or some of the cache partitions but not the entire cache memory, there will be a higher conflict and capacity misses. In fact, the dynamic cache partition re-allocation can help to solve such problem. But, the number of the dynamic cache partition re-allocation is inversely proportional to the task cycle execution time. So, the longer the task cycle execution time, the less number of the dynamic cache partition re-allocation. Finally, the lower compulsory misses can not compensate the increase in the conflict and capacity misses. Therefore, the performance of Model 1 and Model 3 degrade when the task cycle execution time increase.

### 7.1.1 Unified Cache Model Is More Suitable In Real-time Systems

From the simulation results, we observed that unified cache model ( Model 3 ) is more suitable than separate cache model ( Model 1 ) in the real-time computing environment. For separate model, the total number of cache partitions need to be divided by two. Therefore, there will be a keen competition for the cache partition by the tasks. As the instruction stream accessing pattern is more likely in sequential, some of the cache partitions in the instruction cache may be wasted because the task can obtain a higher hit ratio with holding less cache partitions. In contrast, there will be a shortage of cache partitions in the data cache. If we can allocate some of the unused cache partitions in the instruction cache to the data cache, then the data cache hit ratio can increase

undoubtedly. And this situation can be achieved in unified cache model. Although both instruction and data compete for the same place in unified cache, the instruction seems to gather in a cluster rather than scatter across the cache. So, the conflict between the instruction and data can be reduced. Also, as the size of unified cache is doubled than that of separate cache, unified cache suffers from lower capacity and conflict misses than separate cache. Overall, the performance of Model 3 is superior than Model 1.

### 7.1.2 Comments On Aperiodic Tasks

For partitioned cache models, the aperiodic task must have to access the shared cache partition except all the periodic tasks have finished. Whereas for non-partitioned cache models, the aperiodic task will access the whole cache. Although the aperiodic task must suffer from the compulsory misses after preemption, the larger the cache the task used, the lower the conflict and capacity misses. So, generally the aperiodic task in the non-partitioned cache model obtain a high cache hit ratio than that in the partitioned cache model. As a result, the aperiodic task can help to narrow the cache performance gap somehow between partition cache model and non-partitioned cache model.

## 7.2 Future Work

First of all, more program trace files should be used to run on the proposed cache models to get more accurate results. Second, other kinds of task set such as SONAR task set and Avionics task set should be used to verify the performance of the proposed cache models. It is because most of the trace files we used in this simulations are scientific in nature. Therefore, there exists a highly sequentiality in the instruction stream which lead to a high instruction cache hit ratio not only in partitioned cache models but also the non-partitioned ones. Third, other write policies such as write back and no-fetch-on-write can be used to see the performance effects on proposed cache models. Forth, prefetch on hit, prefetch on miss and other kinds of prefetching algorithm such as the software-assisted data prefetching algorithm proposed by Ho[Ho95] can be used to examine the

performance of the proposed cache models.

# Bibliography


[BW89] Alan Burns and Andy Wellings. *Real-Time Systems And Their Programming Languages.* International Computer Science Series, 1989.

[CBM+92] William Y. Chen, Roger A. Bringmann, Scott A. Mahlke, Richard E. Hank, and James E. Sicolo. An Efficient Architecture for Loop Based Data Preloading. In *Conference Proceedings, The 25th International Symposium on Microarchitecture*, pages 92–101, December 1992.

[Cha95] Yung Chan. Multipurpose Short-Term Memory Structures. Master's thesis, Department of Computer Science and Engineering, The Chinese University of Hong Kong, June 1995.

[Che93] Tien-Fu Chen. Data Prefetching for High Performance Processors. Technical report 93-07-01, Department of Computer Science and Engineering, University of Washington, July 1993.

[CS91] Bryce Cogswell and Zary Segall. MACS: A Predictable Architecture for Real Time Systems. In *Proceedings of the Real-Time Systems Symposium*, pages 296–305. IEEE, 1991.

[Das89] Subrata Dasgupda. *Computer Architecture: A Modern Synthesis.* John Wiley And Sons, Inc., 1989.

[ELX88] ELXSI, 2334 Lundy Place, San Jose, CA, 95131. *ELXSI System 6400 - The System Foundation Guide*, 1988.

[Han89] Tom Hand. Real-Time systems need predictability. *Computer Design RISC Supplement*, pages 57–59, August 1989.

[Ho95] Chi-Sum Ho. Software-assisted Data Prefetching. Master's thesis, Department of Computer Science and Engineering, The Chinese University of Hong Kong, June 1995.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publisher, Inc., 3 edition, 1990.

[Hwa93] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, Inc., 1993.

[JDGS91] Dionisios N. Pnevmatikatos Jeffrey D. Gee, Mark D. Hill and Alan Jay Smith. Cache Performance Of The SPEC92 Benchmark Suite. Technical report ucb/csd 91/648, Computer Science Division, University of California at Berkeley, 1991.

[Jou93] Norman P. Jouppi. Cache Write Policies and Performance. In *Conference Proceedings, The 20th International Symposium on Computer Architecture*, pages 191–201, 1993.

[Kir88] David B. Kirk. Process Dependent Static Cache Partitioning for Real-Time Systems. In *Proceedings of the Real-Time Systems Symposium*, pages 181–190, Huntsville, AL, December 1988. IEEE.

[Kir90] David B. Kirk. *Predictable Cache Design for Real-Time Systems.* PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1990.

[KL91] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-controlled Data Prefetching. In *Conference Proceedings, The 18th International Conference on Computer Architecture*, pages 43–53, 1991.

[KS90] David B. Kirk and Jay K. Stronsnider. SMART (Strategic Memory Allocation for Real-Time) Cache Design Using the R3000. In *Proceedings of the Real-Time Systems Symposium*, pages 322–330, Orlando, FA, 1990.

[LL73] C. L. Liu and James. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery 20*, 1:46–61, January 1973.

[MDB88] C. Scheurich M. Dubois and F. A. Briggs. Synchronization, Coherence, and Event Ordering in Multiprocessor. In *IEEE Transaction on Computer*, volume 21(2), pages 9–21, February 1988.

[Oma81] Thabit Khalid Omar. *Cache Management by the Compiler.* PhD thesis, Department of Computer Science, Rice University, May 1981.

[PD95] Fong Pong and Michel Dubois. A New Approach for the Verification of Cache Coherence Protocols. In *IEEE Transactions On Parallel And Distributed Systems*, volume 6, pages 773–787. IEEE, 1995.

[Por89] Alan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercompter Applications.* PhD thesis, Department of Computer Science, Rice University, May 1989.

[PP84] M. Paramarcos and J. Patel. A Low-overhead Coherence solution For Multiprocessors With Private Cache Memories. In *Conference Proceedings, The 11th International Symposium on Computer Architecture*, pages 348–354, June 1984.

[PP94] David A. Patterson and David A. Patterson. *Computer Organization And Design: The Hardware/Software Interface.* Morgan Kaufmann Publisher, Inc., 1994.

[Raj89] Ragunathan Rajkumar. *Task Synchronization in Real-Time Systems.* PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1989.

[RKW94] J. Spencer Love Ramakrishna Karedla and Bradley G. Wherry. Cache Strategies to Improve Disk System Performance. *IEEE Computer*, 27:3:38–46, 1994.

[SLR86] Lehoczky J.P. Sha L. and Rajkumar R. Solutions for some practical problems in prioritizing preemptive scheduling. In *Proceedings IEEE Real-Time Systems Symposium*, 1986.

[Smi82] Alan Jay Smith. Cache Memories. In *Computer Survey*, volume 14, pages 3:473–530, 1982.

[SP95] Jonathan Simonson and Janak H. Patel. Use of Preferred Preemption Points in Cache-Based Real-Time Systems. In *Proceedings of the Real-Time Systems Symposium*, pages 316–325. IEEE, 1995.

[ST86] Harold S. Stone and Dominique F. Thiebaut. Footprints in the Cache. In *Proceeding of the ACM Sigmetrcis Conference on Meas. Mod. of Comp. Sys.*, pages 4–8. ACM, May 1986.